Peer-to-Peer File Sharing System

Gabriel Lancaster-West

**Contents**

[ANALYSIS 12](#_Toc162366439)

[Background to the Problem 12](#_Toc162366440)

[The Current Solution 12](#_Toc162366441)

[Interview with the Client 12](#_Toc162366442)

[What I’ve learned from the interview 13](#_Toc162366443)

[Client Requirements 13](#_Toc162366444)

[Users & Project Limitations 13](#_Toc162366445)

[Possible P2P Solutions 14](#_Toc162366446)

[1. Centralised Index Server 14](#_Toc162366447)

[2. Kademlia 14](#_Toc162366448)

[3. Chord 15](#_Toc162366449)

[Chosen P2P Solution 16](#_Toc162366450)

[Objectives 16](#_Toc162366451)

[Entity Relationship Diagram 17](#_Toc162366452)

[Kademlia 17](#_Toc162366453)

[GUI 18](#_Toc162366454)

[DESIGN 18](#_Toc162366455)

[IPSO Chart 18](#_Toc162366456)

[Validating user information 19](#_Toc162366457)

[Regular Expression testing: 19](#_Toc162366458)

[IP address: 20](#_Toc162366459)

[Constants 20](#_Toc162366460)

[Permanent Storage / Persisting the DHT 21](#_Toc162366461)

[Store it on a peer: 21](#_Toc162366462)

[Store it in a PICKLE file locally: 22](#_Toc162366463)

[Conclusion on choice: 22](#_Toc162366464)

[Creation 22](#_Toc162366465)

[Considerations for Asynchronous code 24](#_Toc162366466)

[Networking 24](#_Toc162366467)

[Processing/Sending Requests 24](#_Toc162366468)

[Algorithms 26](#_Toc162366469)

[Add Contact to Bucket List 26](#_Toc162366470)

[What is it? 27](#_Toc162366471)

[Pseudocode 28](#_Toc162366472)

[Lookup 29](#_Toc162366473)

[What is it? 30](#_Toc162366474)

[Pseudocode (For serial approach) 32](#_Toc162366475)

[K-Bucket Split 33](#_Toc162366476)

[What is it? 34](#_Toc162366477)

[Pseudocode 35](#_Toc162366478)

[Notes on pseudocode: 36](#_Toc162366479)

[Bootstrap 36](#_Toc162366480)

[Pseudocode 36](#_Toc162366481)

[Networking 37](#_Toc162366482)

[Request Handling 37](#_Toc162366483)

[Request Sending 37](#_Toc162366484)

[Handling global variables 37](#_Toc162366485)

[Pseudocode 37](#_Toc162366486)

[ID 37](#_Toc162366487)

[Contact 39](#_Toc162366488)

[KBucket 39](#_Toc162366489)

[Bucket List 40](#_Toc162366490)

[Node 41](#_Toc162366491)

[Server entry points 43](#_Toc162366492)

[Routers 44](#_Toc162366493)

[Base Router 44](#_Toc162366494)

[Router (Inherits Base Router) 44](#_Toc162366495)

[ParallelRouter (Inherits Base Router) 46](#_Toc162366496)

[DHT 46](#_Toc162366497)

[Networking 49](#_Toc162366498)

[Base Server 49](#_Toc162366499)

[HTTP Request Handler 49](#_Toc162366500)

[GUI 50](#_Toc162366501)

[Contact Viewer (Inherits CustomTkinter main window) 50](#_Toc162366502)

[Settings (Inherits CustomTkinter main window) 51](#_Toc162366503)

[Main GUI (Inherits from CustomTkinter main window) 52](#_Toc162366504)

[Join Network Menu Frame (Inherits CustomTkinter frame) 52](#_Toc162366505)

[Upload Frame (Inherits CustomTkinter Frame) 52](#_Toc162366506)

[Download Frame (Inherits CustomTkinter Frame) 53](#_Toc162366507)

[Main Network Frame (Inherits CustomTkinter Frame) 54](#_Toc162366508)

[Load DHT from File Frame (Inherits CustomTkinter Frame) 54](#_Toc162366509)

[Join Network Menu Frame (Inherits CustomTkinter Frame) 54](#_Toc162366510)

[Bootstrap from JSON (Inherits CustomTkinter Frame) 55](#_Toc162366511)

[Bootstrap (Inherits CustomTkinter Frame) 55](#_Toc162366512)

[Class Attributes, Methods and initialisation 56](#_Toc162366513)

[Buckets.py 56](#_Toc162366514)

[Kbucket 56](#_Toc162366515)

[Attributes 56](#_Toc162366516)

[Methods 57](#_Toc162366517)

[BucketList 57](#_Toc162366518)

[Attributes 57](#_Toc162366519)

[Methods 58](#_Toc162366520)

[Constants.py 59](#_Toc162366521)

[Constants 59](#_Toc162366522)

[Contacts.py 59](#_Toc162366523)

[Contact 59](#_Toc162366524)

[Attributes 59](#_Toc162366525)

[Methods 60](#_Toc162366526)

[Dht.py 60](#_Toc162366527)

[DHT 60](#_Toc162366528)

[Attributes 60](#_Toc162366529)

[Methods 61](#_Toc162366530)

[Dictionaries.py 65](#_Toc162366531)

[FindResult(TypedDict) 65](#_Toc162366532)

[ContactQueueItem(TypedDict) 65](#_Toc162366533)

[GetCloserNodesReturn(TypedDict) 66](#_Toc162366534)

[BaseRequest(TypedDict) 66](#_Toc162366535)

[FindNodeRequest(BaseRequest, TypedDict) 66](#_Toc162366536)

[FindValueRequest(BaseRequest, TypedDict) 66](#_Toc162366537)

[PingRequest(BaseRequest, TypedDict) 66](#_Toc162366538)

[StoreRequest(BaseRequest, TypedDict) 66](#_Toc162366539)

[ITCPSubnet(TypedDict) 67](#_Toc162366540)

[FindNodeSubnetRequest(FindNodeRequest, ITCPSubnet, TypedDict) 67](#_Toc162366541)

[FindValueSubnetRequest(FindValueRequest, ITCPSubnet, TypedDict) 67](#_Toc162366542)

[PingSubnetRequest(PingRequest, ITCPSubnet, TypedDict) 67](#_Toc162366543)

[StoreSubnetRequest(StoreRequest, ITCPSubnet, TypedDict) 68](#_Toc162366544)

[CommonRequest(TypedDict) 68](#_Toc162366545)

[BaseResponse(TypedDict) 68](#_Toc162366546)

[ErrorResponse(BaseResponse, TypedDict) 68](#_Toc162366547)

[StoreValue(TypedDict) 68](#_Toc162366548)

[Errors.py 69](#_Toc162366549)

[DataDecodingError (Exception) 69](#_Toc162366550)

[Attributes 69](#_Toc162366551)

[Methods 69](#_Toc162366552)

[TooManyContactsError (Exception) 69](#_Toc162366553)

[Attributes 69](#_Toc162366554)

[Methods 69](#_Toc162366555)

[OutOfRangeError (Exception) 69](#_Toc162366556)

[Attributes 69](#_Toc162366557)

[Methods 69](#_Toc162366558)

[OurNodeCannotBeAContactError (Exception) 70](#_Toc162366559)

[Attributes 70](#_Toc162366560)

[Methods 70](#_Toc162366561)

[AllKBucketsAreEmptyError (Exception) 70](#_Toc162366562)

[Attributes 70](#_Toc162366563)

[Methods 70](#_Toc162366564)

[SendingQueryToSelfError (Exception) 70](#_Toc162366565)

[Attributes 70](#_Toc162366566)

[Methods 70](#_Toc162366567)

[SenderIsSelfError (Exception) 71](#_Toc162366568)

[Attributes 71](#_Toc162366569)

[Methods 71](#_Toc162366570)

[RPCError (Exception) 71](#_Toc162366571)

[Attributes 71](#_Toc162366572)

[Methods 71](#_Toc162366573)

[ValueCannotBeNoneError (Exception) 72](#_Toc162366574)

[Attributes 72](#_Toc162366575)

[Methods 72](#_Toc162366576)

[UnknownRequestError(Exception) 72](#_Toc162366577)

[Attributes 72](#_Toc162366578)

[Methods 72](#_Toc162366579)

[ID.py 72](#_Toc162366580)

[ID 72](#_Toc162366581)

[Attributes 72](#_Toc162366582)

[Methods 73](#_Toc162366583)

[Interfaces.py 74](#_Toc162366584)

[IStorage 74](#_Toc162366585)

[Attributes 74](#_Toc162366586)

[Methods 74](#_Toc162366587)

[IProtocol 76](#_Toc162366588)

[Attributes 76](#_Toc162366589)

[Methods 76](#_Toc162366590)

[Locker.py 76](#_Toc162366591)

[WithLock 77](#_Toc162366592)

[Attributes 77](#_Toc162366593)

[Methods 77](#_Toc162366594)

[My\_queues.py 77](#_Toc162366595)

[InfiniteLinearQueue 77](#_Toc162366596)

[Attributes 77](#_Toc162366597)

[Methods 77](#_Toc162366598)

[LinearQueue 78](#_Toc162366599)

[Attributes 78](#_Toc162366600)

[Methods 78](#_Toc162366601)

[Networking.py 79](#_Toc162366602)

[BaseServer(ThreadingHTTPServer) 79](#_Toc162366603)

[Attributes 79](#_Toc162366604)

[Methods 79](#_Toc162366605)

[HTTPSubnetRequestHandler(BaseHTTPRequestHandler) 80](#_Toc162366606)

[Attributes 80](#_Toc162366607)

[Methods 80](#_Toc162366608)

[TCPSubnetServer(BaseServer) 80](#_Toc162366609)

[Attributes 80](#_Toc162366610)

[Methods 81](#_Toc162366611)

[HTTPRequestHandler(BaseHTTPRequestHandler) 81](#_Toc162366612)

[Attributes 81](#_Toc162366613)

[Methods 81](#_Toc162366614)

[TCPServer(BaseServer) 82](#_Toc162366615)

[Attributes 82](#_Toc162366616)

[Methods 82](#_Toc162366617)

[Node.py 82](#_Toc162366618)

[Node 82](#_Toc162366619)

[Attributes 82](#_Toc162366620)

[Methods 83](#_Toc162366621)

[Protocols.py 85](#_Toc162366622)

[VirtualProtocol(IProtocol) 85](#_Toc162366623)

[Attributes 85](#_Toc162366624)

[Methods 86](#_Toc162366625)

[TCPSubnetProtocol(IProtocol) 87](#_Toc162366626)

[Attributes 87](#_Toc162366627)

[Methods 87](#_Toc162366628)

[TCPProtocol(IProtocol) 89](#_Toc162366629)

[Attributes 89](#_Toc162366630)

[Methods 90](#_Toc162366631)

[Routers.py 92](#_Toc162366632)

[BaseRouter 92](#_Toc162366633)

[Attributes 92](#_Toc162366634)

[Methods 93](#_Toc162366635)

[Router(BaseRouter) 94](#_Toc162366636)

[Attributes 94](#_Toc162366637)

[Methods 95](#_Toc162366638)

[ParallelRouter(BaseRouter) 95](#_Toc162366639)

[Attributes 95](#_Toc162366640)

[Methods 96](#_Toc162366641)

[Storage.py 98](#_Toc162366642)

[VirtualStorage(IStorage) 99](#_Toc162366643)

[Attributes 99](#_Toc162366644)

[Methods 99](#_Toc162366645)

[SecondaryJSONStorage(IStorage) 100](#_Toc162366646)

[Attributes 100](#_Toc162366647)

[Methods 100](#_Toc162366648)

[GUI.py 101](#_Toc162366649)

[Fonts 102](#_Toc162366650)

[Attributes 102](#_Toc162366651)

[ContactViewer(customtkinter.CTk) 102](#_Toc162366652)

[Attributes 102](#_Toc162366653)

[Methods 102](#_Toc162366654)

[StatusWindow (customtkinter.CTk) 103](#_Toc162366655)

[Attributes 103](#_Toc162366656)

[Methods 103](#_Toc162366657)

[Settings (customtkinter.CTk) 103](#_Toc162366658)

[Attributes 103](#_Toc162366659)

[Methods 104](#_Toc162366660)

[ErrorWindow (customtkinter.CTk) 105](#_Toc162366661)

[Attributes 105](#_Toc162366662)

[Methods 105](#_Toc162366663)

[MainGUI (customtkinter.CTk) 105](#_Toc162366664)

[Attributes 105](#_Toc162366665)

[Methods 105](#_Toc162366666)

[UploadFrame(customtkinter.CTkFrame) 108](#_Toc162366667)

[Attributes 108](#_Toc162366668)

[Methods 108](#_Toc162366669)

[DownloadFrame(customtkinter.CTkFrame) 109](#_Toc162366670)

[Attributes 109](#_Toc162366671)

[Methods 109](#_Toc162366672)

[MainNetworkFrame(customtkinter.CTkFrame) 111](#_Toc162366673)

[Attributes 111](#_Toc162366674)

[Methods 111](#_Toc162366675)

[LoadDHTFromFileFrame(customtkinter.CTkFrame) 111](#_Toc162366676)

[Attributes 111](#_Toc162366677)

[Methods 112](#_Toc162366678)

[JoinNetworkMenuFrame(customtkinter.CTkFrame) 113](#_Toc162366679)

[Attributes 113](#_Toc162366680)

[Methods 113](#_Toc162366681)

[BootstrapFromJSON(customtkinter.CTkFrame) 113](#_Toc162366682)

[Attributes 113](#_Toc162366683)

[Methods 114](#_Toc162366684)

[BootstrapFrame(customtkinter.CTkFrame) 115](#_Toc162366685)

[Attributes 115](#_Toc162366686)

[Methods 116](#_Toc162366687)

[Node Lookup 117](#_Toc162366688)

[Creating Kademlia 118](#_Toc162366689)

[User Interface Design 118](#_Toc162366690)

[Storing Files 119](#_Toc162366691)

[TECHNICAL SOLUTION 122](#_Toc162366692)

[What techniques have I used? 122](#_Toc162366693)

[Reviewing Objectives 123](#_Toc162366694)

[TESTING 132](#_Toc162366695)

[Debugging 132](#_Toc162366696)

[Problem 1 - Buckets split incorrectly 132](#_Toc162366697)

[Problem 2 – No closest contacts are returned 135](#_Toc162366698)

[Problem 3 – Unsorted close contacts 139](#_Toc162366699)

[Problem 4 – Bootstrapping Issues 139](#_Toc162366700)

[Ensuring decode/encode functions correctly 144](#_Toc162366701)

[Networking tests 145](#_Toc162366702)

[Ping test 146](#_Toc162366703)

[DHT not saving correctly due to pickle 150](#_Toc162366704)

[GUI Tests 151](#_Toc162366705)

[Could I make it save to somewhere other than the project directory? 152](#_Toc162366706)

[Can I make it save to a folder which does not exist? 153](#_Toc162366707)

[Our contact not saving correctly 156](#_Toc162366708)

[Bootstrapped contact not responding 159](#_Toc162366709)

[Downloading 161](#_Toc162366710)

[Unit Test Tables 162](#_Toc162366711)

[Input validation test tables 163](#_Toc162366712)

[Unit test return 167](#_Toc162366713)

[EVALUATION 169](#_Toc162366714)

[Client Feedback 169](#_Toc162366715)

[Scope for further development 169](#_Toc162366716)

[Comparison of achieved solution versus Objectives 170](#_Toc162366717)

[Have a system which can send data between 2 devices using IP and port number. 170](#_Toc162366718)

[Use some method to determine which device has the requested data 170](#_Toc162366719)

[Query devices asynchronously. 170](#_Toc162366720)

[GUI to coordinate the transfer process 171](#_Toc162366721)

[Must be scalable and lightweight. 171](#_Toc162366722)

[Should be aesthetically pleasing. 171](#_Toc162366723)

[APPENDIX 172](#_Toc162366724)

[Code 172](#_Toc162366725)

[Buckets.py 172](#_Toc162366726)

[Constants.py 178](#_Toc162366727)

[Contact.py 179](#_Toc162366728)

[dht.py 179](#_Toc162366729)

[dictionaries.py 191](#_Toc162366730)

[errors.py 194](#_Toc162366731)

[helpers.py 197](#_Toc162366732)

[id.py 199](#_Toc162366733)

[interfaces.py 202](#_Toc162366734)

[locker.py 205](#_Toc162366735)

[my\_queues.py 206](#_Toc162366736)

[networking.py 208](#_Toc162366737)

[node.py 216](#_Toc162366738)

[pickler.py 222](#_Toc162366739)

[protocols.py 223](#_Toc162366740)

[routers.py 237](#_Toc162366741)

[storage.py 251](#_Toc162366742)

[GUI.py 258](#_Toc162366743)

[Unit tests 275](#_Toc162366744)

[Test References 312](#_Toc162366745)

[THE END. 356](#_Toc162366746)

# ANALYSIS

## Background to the Problem

My friend Dan wants to share files between multiple computers. Dan is quite paranoid and doesn’t want to store his data under the prying eyes of large corporations. There should be capabilities for multiple (more than two) computers to share files, in case Dan adds more computers, or wants to share files with his friends.

## The Current Solution

Currently, my client Dan, is backing his folders up to Microsoft OneDrive, which stores a copy of folders on a central server. However, without paying, the standard user only has 500MB of storage, which isn’t enough to sync large videos, or basic apps.

A screenshot of a computer

Description automatically generated

*Above: Current system – Microsoft OneDrive.*

To share between different computers, Dan is having to sign into OneDrive on each device to access his profile on the central server, then he is manually downloading them from the website.

## Interview with the Client

This is a message chain I had with my client before beginning my project – when referencing the “current system”, we’re referring to Microsoft OneDrive at <https://onedrive.live.com/>:

Me: “What do you like about the current system?”

Dan: “it’s simple to add new devices”

Me: “What do you not like about it?”

Dan: “It’s slow, it costs tons for storage, and its run by a massive corporation so they might be able to see all my stuff”

Me: “What could be improved with it?”

Dan: “Small or lighter files to run on computers (Not as important), and the storage should be cheaper or free”

Me: “What do you mean by small or lighter files?”

Dan: “Less computationally intense/more lightweight”

Me: “Do you like the look of it?”

Dan: “Could do with a dark mode I guess but apart from that yeah”

### What I’ve learned from the interview

* My client likes how easy it is to add new devices to the current system.
* My client wants to be able to run the current system on a computer with worse hardware.
* He dislikes the security of the current solution, because it is vulnerable to Microsoft, because it is still stored in their servers.
* A financially cheap solution is very important for my client.
* My client likes the idea of having a dark mode, but he doesn’t seem to care too much about the aesthetics of the project – I probably could get away with a CLI and he’d be happy, but there’s no way I would be happy with that.

### Client Requirements

1. The program must have mostly the same base elements as the current system; this includes the ability to download and open files from the UI/GUI.
2. The program must be easy to add devices to.
3. The overall system must be relatively lightweight.
4. The data must not be stored in an area where there is a risk that big companies can read your data – so it should not be backed up onto GitHub or added to OneDrive/Google Drive.
5. Must cost as little as possible – so preferably not using a server to avoid electricity costs.
6. The program should be in dark mode.

## Users & Project Limitations

Although I only have a single primary client, I would like to make this system capable to be used by many people to share data across the world on separate networks. My client is extremely proficient with technology; however, I want to make this sufficiently user friendly, so that it is reasonably intuitive as to how it works to a new user. Because of this, it is unlikely that a command line interface will be used.

The limitations to this project are the following:

* What I know: I am fluent in Python, and proficient in Rust & PHP. However, I’m not familiar with many other languages – such as Java, JavaScript, C, or C++; I am a fast learner though, so I could learn them if necessary for this project.
* Time: I have until February/March 2024 to complete this project, giving me 5 months to complete this from start to finish.
* Python: Python can do almost anything another language can, with the help of external modules, which require pip to install most of the time.
* My school: My organisation’s computers do not allow users to add executables to the windows PATH environment variable, which may cause pip to not work – therefore external modules may prove a challenge. My organisation also sometimes removes all SSL certificates (or something of this nature), which does not allow pip or git to function as intended out of the box, which makes working on the code a massive problem while at school.
* Databases: I’m not fully comfortable with SQL, having knowledge of basic commands from lessons, but I have no real world experience with them. However, it seems fairly self-explanatory so I could definitely learn this if I needed to.

## Possible P2P Solutions

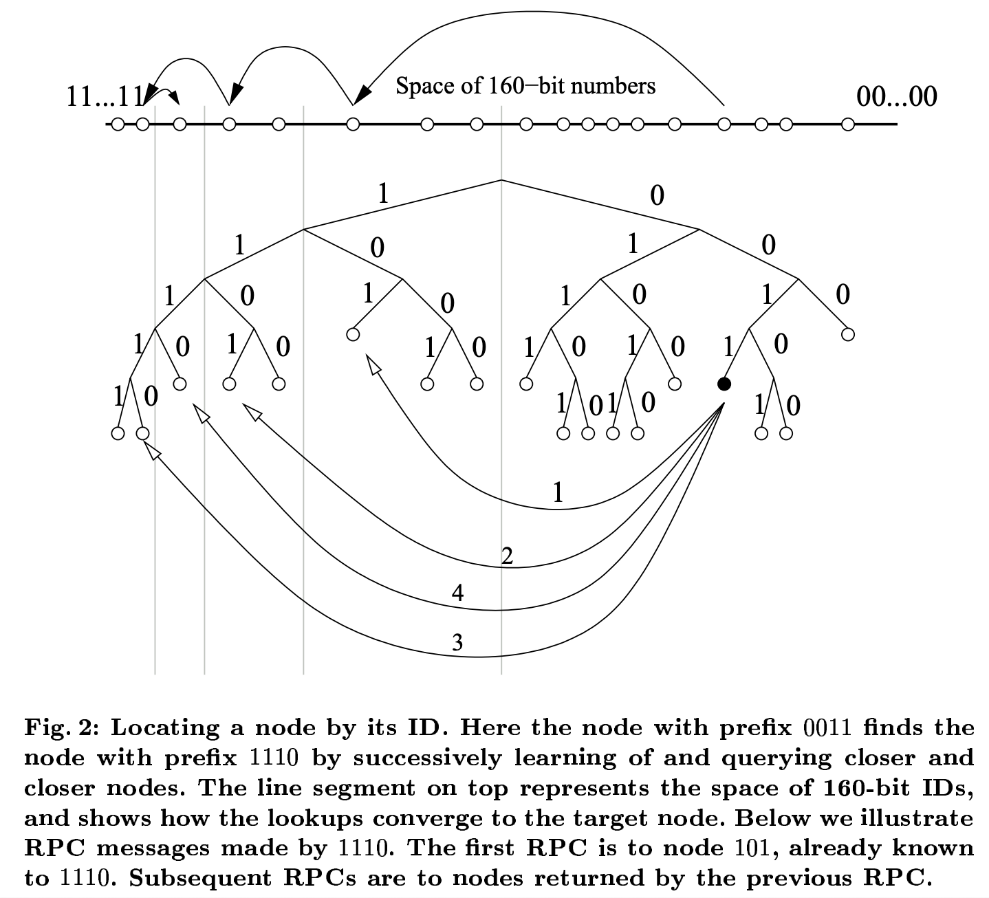
### Centralised Index Server

I could make a system which uses a central index server to coordinate data transfer between devices, like Gnutella. It should be noted here that this is still peer-to-peer, and the central server is used for device discovery. As there is a central trusted server, I could use this to coordinate HTTPS encryption between devices.

* If the server goes down, the whole network goes down.
* My client will have to rent a server, which costs money.
* It would be very easy to have a graphical interface of all devices and what they store, as that is all stored on the central server using something such as SQL.

### Kademlia

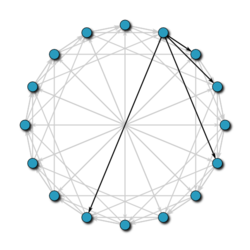
A system which uses the Kademlia Protocol could be created. Kademlia a very efficient peer-to-peer algorithm, to locate devices on an overlay network. Kademlia uses UDP to exchange data through ‘node lookups’ where each node is identified using a node ID (a 160-bit positive binary integer in the original specification), the Kademlia algorithm uses this ID to locate values – such as file hashes or keywords.



* This does not require a central server, so the network will always work on a device, unless all the devices “k-buckets” are offline, which is configured to practically never happen in a sufficiently large network.
* As there is no central server, HTTPS cannot be used. I believe that SFTP (SSH File Transfer Protocol) would work, as it relies on SSH credentials to symmetrically securely send data.
* A recursive node lookup would have to be used to show all devices and what they store, I may have to create an SQL database for user lookups by an individual device, but this may be impractical with large networks.

### Chord

I could create a system which uses Chord to locate devices on a network. It would then use a separate file transfer protocol, such as PGP (Pretty Good Privacy), which uses a mix of symmetric and asymmetric encryption to transfer data. In chord, each node has a ‘finger table’ to avoid a linear search for nodes. This table is such that the *ith* entry of node *n* will contain . This creates an order of , like Kademlia.

1. This also does not require a central server and should never go offline if there are online nodes.
2. In order to show all files available, a recursive node lookup would be required, and for easy use filtering I would use a list of dictionaries in Python, this would be impractical however for large networks, and if this is true this would be swapped to being stored in a JSON file.

*Above: A 16 node network. The ‘finger table’ for one of the nodes is in black.*

## Chosen P2P Solution

I will use the peer-to-peer system ‘Kademlia’. I shall use Kademlia primarily for its simplicity in handling offline devices when compared to Chord, as Kademlia’s load bearing is much more flexible than Chord, because it dynamically routes itself, instead of relying on a consistent hashing technique.

## Objectives

What is needed in this project:

1. Have a system which can send data between 2 devices using IP and port number.
2. Use some method to determine which device has the requested data – I have chosen to implement the Kademlia protocol here. It must have the following traits:
   * I must have a functioning implementation of “KBuckets”, these are the lists of ID’s that each device knows about.
     + Each must split properly and at the correct time.
     + Each KBucket should have a range with high and low boundaries on powers of 2.
     + Contacts are added to the correct KBucket from the bucket list.
   * Create server to handle incoming requests.
   * Unresponsive nodes should be evicted from the network.
   * Method to lookup nodes on the network.
   * Method to find nodes on the network.
   * (Optional) ID object which can be used interchangeably with integers.
   * Ability to send RPC (Remote Procedure Call) requests to other nodes on the network

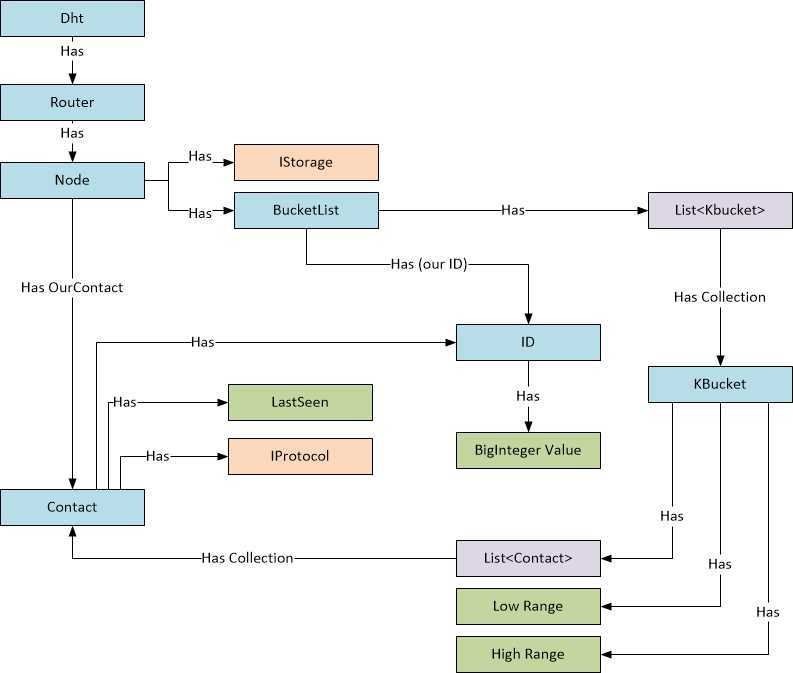
1. Query devices asynchronously.
2. GUI to coordinate the transfer process, either an app made in TKinter/customTkinter or a webapp made in Flask or React, like that used in many similar software.
3. Must be scalable and lightweight.

Should be aesthetically pleasing.

1. The GUI should have a dark mode.

## Entity Relationship Diagram

### Kademlia



This is taken from an explanation of the Kademlia Protocol (See “The Kademlia Protocol Succinctly” - Getting Started – A framework for the implementation, available at <https://www.syncfusion.com/succinctly-free-ebooks/kademlia-protocol-succinctly/getting-started>).

### GUI

A computer screen shot of a diagram

Description automatically generated

This is the entity relationship diagram for my GUI. The diamond in the centre represents the method which initialises a new Kademlia network; although it is not an entity, it is a substantial part of the GUI process that I believe it should be included, so that it will keep making sense.

# DESIGN

## IPSO Chart

|  |  |
| --- | --- |
| INPUT | PROCESS |
| Bootstrap:   * IP address of known peer * Port of known peer * ID of known peer   Bootstrap from JSON:   * Path to file containing known peer   Upload file:   * Path to file   Download file:   * ID of file | Bootstrap:   * Validate IP * Validate port * Validate ID * Make new DHT object * Bootstrap DHT object using known peer   Bootstrap from JSON:   * Ensure file exists * Validate file * Validate contents (IP, port, ID) * Make new DHT object * Bootstrap DHT object using known peer   Upload file:   * Ensure there is a file at the given path * Create dictionary containing filename and file data, and upload pickled dictionary to storage   Download file:   * Send GET\_VALUE RPC across the network * Download file |
| STORAGE | OUTPUT |
| * Key-Value pairs of {ID: File data} | * Our contact (as JSON) * Our DHT object (as PICKLE) * Downloaded files |

## Validating user information

There will be many user inputs into the system; here is how I will deal with each:

|  |  |
| --- | --- |
| User input | Validation |
| IP Address | Must be either “localhost” or 4 integers (up to 255) separated by full stops. |
| Regular Expression:  (([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5]) |
| Port number | Must be an integer between 0 and 65535. This can be |
| File path to upload | Must be a valid path to an individual file; this can be checked using os.path.isfile() |
| ID | Must be an integer between 0 and 2160. This can be checked using |

### Regular Expression testing:

#### IP address:

A screenshot of a computer

Description automatically generated

I used an [online regex checker](https://regex101.com/) to check my regular expressions.

This correctly excludes 4 full stops, and anything containing negative numbers. However, it excludes IPs in the form “A”, “A.B”, “A.B.C”, which are extremely uncommon, so I do not believe this to be an issue – I was not aware that they existed before this.

IP addresses will be validated using the Python Regex module “re”, which is built in.

**Breakdown of the regular expression:**

(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])

* “[0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5]” Checks for all integers between 0 and 255.
* “\.” Checks that a full stop follows.
* “{3}” matches the previous token 3 times.

## Constants

Kademlia uses a variety of contacts, which are not explicitly stated in the original paper. Many people have different interpretations of these constants.

I am going to create the following constants:

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| K | 20 | Size of a k-bucket. |
| B | 5 | Used to limit a nodes understanding of a network; it is used to check if a k-bucket can split. |
| A | 20 | How many nodes should be queried when lookup is called. |
| REQUEST\_TIMEOUT | 0.5 | How long before a TimeoutError is raised when attempting to contact an unresponsive node. |
| ID\_LENGTH\_BYTES | 20 | Length of an ID in bytes. |
| ID\_LENGTH\_BITS | 160 | Length of an ID in bits. |
| MAX\_THREADS | 20 | Number of threads created in a ParallelRouter. |
| RESPONSE\_WAIT\_TIME | 10 | Time to sleep in milliseconds between lookup calls to wait for a response. |
| BUCKET\_REFRESH\_INTERVAL | 3,600,000 | (Once per hour) How often buckets should be refreshed in milliseconds. |
| KEY\_VALUE\_REPUBLISH\_INTERVAL | 3,600,000 | (Once per hour) How often key-value pairs should be republished in milliseconds. |
| KEY\_VALUE\_EXPIRE\_INTERVAL | 3,600,000 | (Once per hour) how long before a key-value pair expires in milliseconds. |
| ORIGINATOR\_REPUBLISH\_INTERVAL | 86,400,000 | (Once per day) How often originator storage should update in milliseconds. |
| EXPIRATION\_TIME\_SEC | 86,400 | (Once per day) How often cache storage should update in seconds. |
| EVICTION\_LIMIT | 5 | How many attempts to contact a node before they are evicted from the network. |
| DEBUG | False | Whether or not debugging should be enabled – this simplifies some functions and sets a seed for randomness for consistent testing. |
| TRY\_CLOSEST\_BUCKET | True | If the lookup algorithm should look in the closest bucket initially. |

## Permanent Storage / Persisting the DHT

The bucket lists must not be volatile, so there must be a way to store this data. There are several ways to do this:

### Store it on a peer:

I could store our bucket lists on another peer on the network, and then fetch it upon restart. This feels like it would fit with the idea of a peer-to-peer system, but it is entirely unnecessary and creates unnecessary security risks; The peer that this is stored on can edit the data – unless it was encrypted, but the peer that this is stored on will still have to be stored, rendering this method near useless, as something would still have to be stored on our computer.

### Store it in a PICKLE file locally:

Python’s library “pickle” is great for storing objects, however I am not sure if it can handle circular references, such as “self.node.DHT = self”, to allow nodes to view and access superior classes, which they are not inheriting from.

*import* pickle  
  
example\_dict = {1:"6",2:"2",3:"f"}  
  
pickle\_out = open("dict.pickle","wb")  
pickle.dump(example\_dict, pickle\_out)  
pickle\_out.close()  
  
pickle\_in = open("dict.pickle","rb")  
example\_dict = pickle.load(pickle\_in)

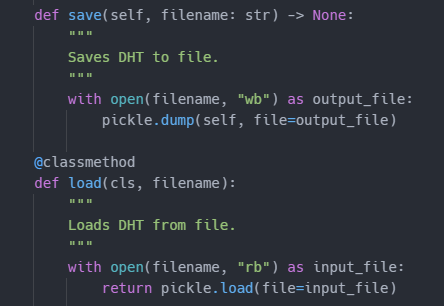
*Above: Example usages of pickle from* [*here.*](https://pythonprogramming.net/python-pickle-module-save-objects-serialization/#:~:text=First%2C%20import%20pickle%20to%20use,into%20opened%20file%2C%20then%20close.)

Note: Apparently it can, according to a [StackOverflow thread,](https://stackoverflow.com/questions/7433963/can-pickle-handle-multiple-object-references#:~:text=Yes%2C%20shared%20objects%20will%20only,won't%20be%20serialized%20again.) which I cannot access due to my schools network restrictions. But the Google preview states the following:  
“Yes, shared objects will only get serialized once (the pickle protocol can even handle circular references). From the documentation: The pickle module keeps track of the objects it has already serialized, so that later references to the same object won't be serialized again.” - 15 Sept 2011

### Conclusion on choice:

I think I will store it using the pickle library because it seems to be well suited for circular referencing.

### Creation



*Above: Initial implementation of DHT serialisation.*

I can now write some tests for this:

1. class DHTSerialisationTests(unittest.TestCase):

2. def test\_serialisation(self):

3. dht: DHT = DHT(

4. id=ID.random\_id(),

5. protocol=VirtualProtocol(),

6. router=Router(),

7. storage\_factory=VirtualStorage

8. )

9. dht.save("dht.pickle")

10.

11. new\_dht = DHT.load("dht.pickle")

12.

13. self.assertTrue(

14. type(dht) == type(new\_dht),

15. "Saved and loaded DHT are not the same type. " \

16. f"{type(dht)} vs {type(new\_dht)}"

17. )

18. self.assertTrue(

19. dht.our\_id == new\_dht.our\_id,

20. "Saved and loaded DHT is not identical to the original."

21. )

22.

23. def test\_circular\_serialisation(self):

24. dht: DHT = DHT(

25. id=ID.random\_id(),

26. protocol=VirtualProtocol(),

27. router=Router(),

28. storage\_factory=VirtualStorage

29. )

30.

31. node = Node(

32. Contact(dht.our\_id),

33. storage=VirtualStorage()

34. )

35. dht.\_router.node = node

36.

37. dht.save("dht.pickle")

38.

39. new\_dht = DHT.load("dht.pickle")

40.

41. self.assertTrue(

42. type(dht) == type(new\_dht),

43. "Saved and loaded DHT are not the same type. " \

44. f"{type(dht)} vs {type(new\_dht)}"

45. )

46.

47. self.assertTrue(

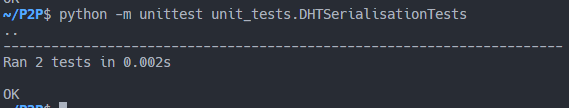
48. dht.\_router.node.our\_contact.id == new\_dht.\_router.node.our\_contact.id,

49. "Saved and loaded DHT is not identical to the original."

50. )

51.

*Above: DHT Serialisation Tests*



*Above: Testing results*

It worked first try!

## Considerations for Asynchronous code

Collections should not be modified or changed in any way while they are being searched, so bucket lists should be locked while they are being searched.

RPCs must be sent in parallel according to the specification:

“The initiator then sends parallel, asynchronous find\_node RPCs to the ‘a’ nodes it has chosen, a is a system-wide concurrency parameter, such as 3.”

This is done to reduce the impact of timeout delays from failed nodes.

This seems incredibly useful and will increase the effectiveness of my solution. However, it is very complex, and I have never worked with asynchronous programs, “asyncio” (the main Python asynchronous library) before, or “threading” (the main python library for utilizing threads).

This may be necessary when handling requests, but I’m not sure.

## Networking

### Processing/Sending Requests

I will use the http.server library for this. I believe “socket” is too simple for this, and I will have to create a lot of HTTP from the ground up / or end up making a bad version of HTTP, which I don’t understand the point of. The http.server library has, what I believe to be the perfect object for me, which is HTTPServer/HTTPThreadingServer, has a handlers such as “BaseHTTPRequestHandler”, which has some great methods such as handle\_one\_request(), which handles a single HTTP request.



This warning is concerning, as it goes against my “secure transfer” ideology – even though I plan to encrypt the data anyway.

A screenshot of a computer

Description automatically generated

However, as long as I use Python 3.12 or above, this seems to be irrelevant.

A screen shot of a computer

Description automatically generatedA computer screen shot of a program

Description automatically generatedA quick check of an HTTPServer() demo, that I found on the internet shows that it works:

I want to handle HTTP “POST” requests sent to the client. This library appears to do that for me. After handling the request, it should save all the information to the following:

|  |  |  |
| --- | --- | --- |
| Useful Attribute | Type | Information |
| command | String | 'Contains the command (request type). For example, "GET".' |
| client\_address | Tuple | 'Contains a tuple of the form (host, port) referring to the client’s address.' |
| path | string | 'Contains the request path. If query component of the URL is present, then path includes the query. Using the terminology of RFC 3986, path here includes hier-part and the query.' |
| headers | [email.message.Message](https://docs.python.org/3/library/email.message.html#email.message.EmailMessage) (see [email.message: Representing an email message — Python 3.12.2 documentation](https://docs.python.org/3/library/email.message.html#email.message.EmailMessage)) | 'Holds an instance of the class specified by the MessageClass class variable. This instance parses and manages the headers in the HTTP request. The parse\_headers() function from http.client is used to parse the headers and it requires that the HTTP request provide a valid RFC 2822 style header.' |
| rfile | io.BufferedIOBase | ‘An io.BufferedIOBase input stream, ready to read from the start of the optional input data.’ |

Further attribute information can be found in [the Python “http.server” documentation](https://docs.python.org/3.12/library/http.server.html). All information in quotes is taken from that.

To parse the headers into a standard dictionary, I should just be able to perform:

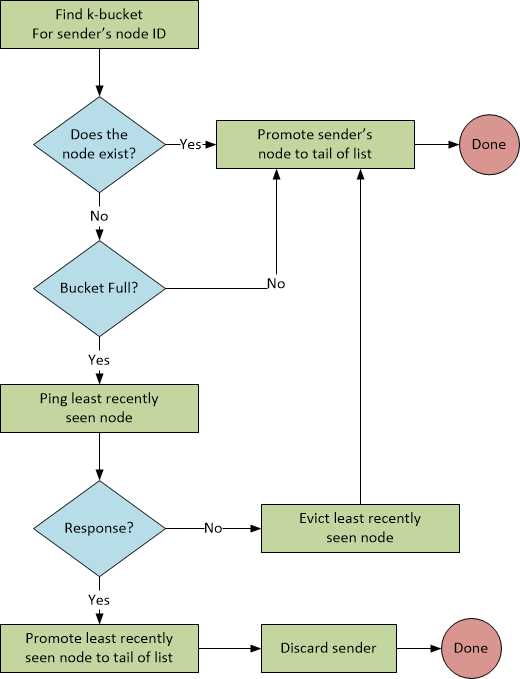
These headers can contain crucial information such as which subnet should be used for unit testing. However, headers typically have a standard size – although that does not matter here – so I don’t think an entire HTTP request should be sent in the headers.

So I should use json.dumps() to dump a dictionary to string, and I could pickle all elements of the dictionary, so then their class information can be retained when received.

## Algorithms

### Add Contact to Bucket List

#### What is it?



*Above: “The Add Contact Algorithm” – The Kademlia Protocol Succinctly, Marc Clifton*

The original specification says…

“When a Kademlia node receives any message (request or reply) from another node, it updates the appropriate k-bucket for the sender’s node ID. If the sending node already exists in the recipient’s k-bucket, the recipient moves it to the tail of the list. If the node is not already in the appropriate k-bucket and the bucket has fewer than k entries, then the recipient just inserts the new sender at the tail of the list. If the appropriate k-bucket is full, however, then the recipient pings the k-bucket’s least-recently seen node to decide what to do. If the least recently seen node fails to respond, it is evicted from the k-bucket and the new sender inserted at the tail. Otherwise, if the least-recently seen node responds, it is moved to the tail of the list, and the new sender’s contact is discarded”.

The ‘appropriate k-bucket’ is the k-bucket which has the sender’s node ID within its range.

#### Pseudocode

SUB add\_contact(self, contact):

IF self.our\_id == contact.id:

RAISE OurNodeCannotBeAContactError("Cannot add ourselves as a contact.")

ENDIF

contact.touch()

kbucket: KBucket = self.get\_kbucket(contact.id)

IF kbucket.contains(contact.id):

OUTPUT "Contact already in KBucket."

// replace contact, then touch it

kbucket.replace\_contact(contact)

ELSE IF kbucket.is\_full():

IF self.can\_split(kbucket):

// Split then try again

k1, k2 = kbucket.split()

index = self.\_get\_kbucket\_index(contact.id)

// adds the two buckets to 2 separate buckets.

self.buckets[index] = k1 // Replaces original KBucket

self.buckets.insert(index + 1, k2) // Adds a new one after it

self.add\_contact(contact) // Unless k <= 0, This should never cause a recursive loop

ELSE:

last\_seen\_contact = SORT\_LIST\_ASC(kbucket.contacts, key=lambda c: c.last\_seen)[0]

error = last\_seen\_contact.protocol.ping(self.our\_contact)

IF error != None:

IF self.DHT != None: // tests may not initialise a DHT

self.DHT.delay\_eviction(last\_seen\_contact, contact)

ENDIF

ELSE

// still can't add the contact, so put it into the pending list

IF self.DHT != None:

self.DHT.add\_to\_pending(contact)

ENDIF

ENDIF

ENDIF

ELSE:

// Bucket is not full, nothing special happens.

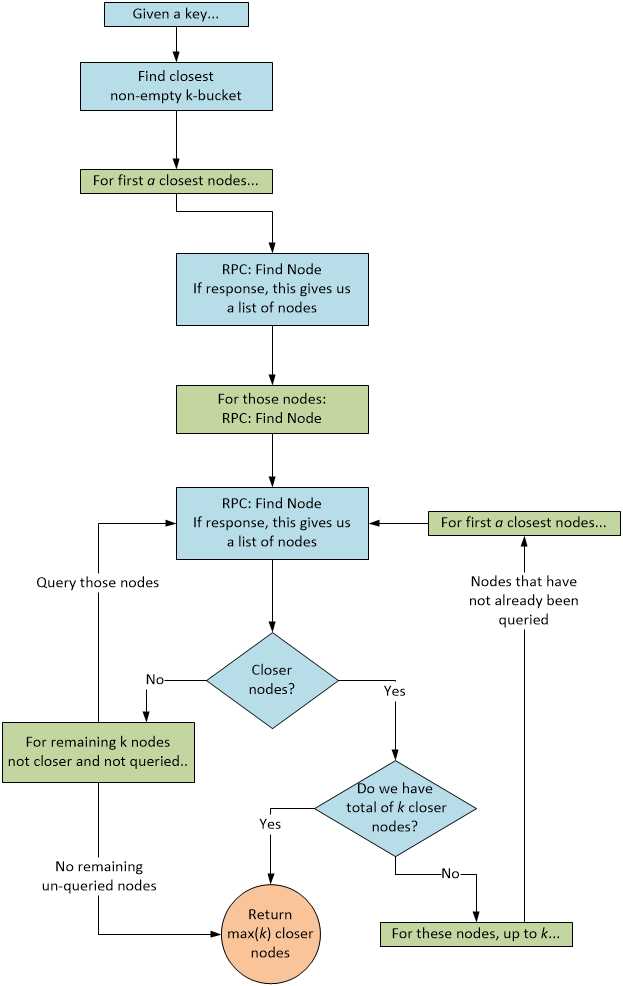
kbucket.add\_contact(contact)

ENDIF

ENDSUB

### Lookup

#### What is it?



*Above: “Node Lookup Algorithm” – The Kademlia Protocol Succinctly*

The specification says:

“The most important procedure a Kademlia participant must perform is to locate the k closest nodes to some given node ID. We call this procedure a node lookup. Kademlia employs a recursive algorithm for node lookups. The lookup initiator starts by picking a nodes from its closest non-empty k-bucket (or, if that bucket has fewer than a entries, it just takes the a closest nodes it knows of). The initiator then sends parallel, asynchronous FIND\_NODE RPCS to the a nodes it has chosen, a is a system-wide concurrency parameter, such as 3… …In the recursive step, the initiator resends the FIND\_NODE to nodes it has learned about from previous RPCs. (This recursion can begin before all a of the previous RPCs have returned). Of the k nodes the initiator has heard of closest to the target, it picks a that it has not yet queried and resends the FIND\_NODE RPC to them. Nodes that fail to respond quickly are removed from consideration until and unless they do respond. If a round of FIND\_NODES fails to return a node any closer than the closest already seen, the initiator resends the FIND\_NODE to all of the k closest nodes it has not already queried. The lookup terminates when the initiator has queried and gotten responses from the k closest nodes it has seen. When a = 1, the lookup algorithm resembles Chord’s in terms of message cost and the latency of detecting failed nodes. However, Kademlia can route for lower latency because it has the flexibility of choosing any one of k nodes to forward a request to.”

A simple explanation is that we search for nodes closest to the given key. This starts by getting nodes from the closest non-empty k-bucket and continues querying nodes serially with a given RPC call until it has gathered responses from the closest k nodes or until we run out of nodes to contact. This takes a lot of time, so there is also the parallel approach, which would have to use threads and semaphores to handle a large number of contacts at once, and to prevent data being corrupted.

#### Pseudocode (For serial approach)

1. SUB lookup(self, key, rpc\_call):

2. all\_nodes = self.node.bucket\_list.get\_close\_contacts(key, self.node.our\_contact.id)[0:Constants.K]

3.

4. nodes\_to\_query = all\_nodes[0:Constants.A]

5.

6. FOR i IN nodes\_to\_query:

7. IF i.id.value ^ key.value < self.node.our\_contact.id.value ^ key.value:

8. self.closer\_contacts.append(i)

9. ELSE:

10. self.further\_contacts.append(i)

11. ENDIF

12. ENDFOR

13.

14. // all untested contacts just get dumped here.

15. FOR i IN all\_nodes[Constants.A + 1:]:

16. self.further\_contacts.append(i)

17. ENDFOR

18.

19. FOR i IN nodes\_to\_query:

20. IF i NOT IN contacted\_nodes:

21. contacted\_nodes.append(i)

22. ENDIF

23. ENDFOR

24.

25. # In the spec they then send parallel async find\_node RPC commands

26. query\_result = self.query(key, nodes\_to\_query, rpc\_call, self.closer\_contacts, self.further\_contacts)

27.

28. IF query\_result["found"]: # if a node responded

29. RETURN query\_result

30. ENDIF

31.

32. # add any new closer contacts

33. FOR i IN self.closer\_contacts:

34. IF i.id NOT IN [j.id FOR j IN ret]: # if id does not already exist inside list

35. ret.append(i)

36. ENDIF

37. ENDFOR

38.

39. WHILE len(ret) < Constants.K AND have\_work:

40. closer\_uncontacted\_nodes = [i FOR i IN self.closer\_contacts IF i NOT IN contacted\_nodes]

41. further\_uncontacted\_nodes = [i FOR i IN self.further\_contacts IF i NOT IN contacted\_nodes]

42.

43. # If we have uncontacted nodes, we still have work to be done.

44. have\_closer = LENGTH(closer\_uncontacted\_nodes) > 0

45. have\_further = LENGTH(further\_uncontacted\_nodes) > 0

46. have\_work = have\_closer OR have\_further

47.

48. IF have\_closer:

49. new\_nodes\_to\_query = closer\_uncontacted\_nodes[:Constants.A]

50. FOR i IN new\_nodes\_to\_query:

51. IF i NOT IN contacted\_nodes:

52. contacted\_nodes.append(i)

53. ENDIF

54. ENDFOR

55.

56. query\_result = self.query(key, new\_nodes\_to\_query, rpc\_call, self.closer\_contacts, self.further\_contacts)

57.

58. IF query\_result["found"]:

59. RETURN query\_result

60. ENDIF

61.

62. ELSE IF have\_further:

63. new\_nodes\_to\_query = further\_uncontacted\_nodes[:Constants.A]

64. FOR i IN new\_nodes\_to\_query:

65. IF i NOT IN contacted\_nodes:

66. contacted\_nodes.append(i)

67. ENDIF

68. ENDFOR

69.

70. query\_result = self.query(key, new\_nodes\_to\_query, rpc\_call, self.closer\_contacts, self.further\_contacts)

71.

72. IF query\_result["found"]:

73. RETURN query\_result

74. ENDIF

75. ENDIF

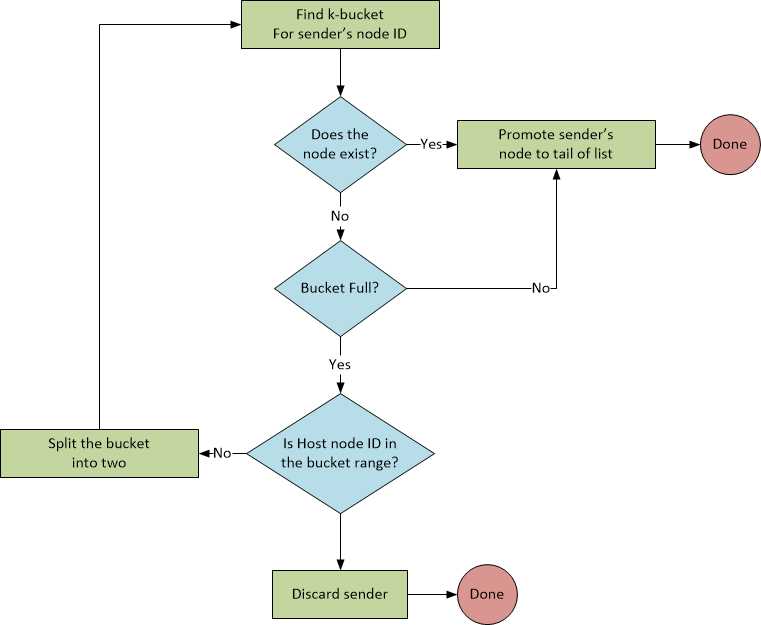
76. ENDWHILE

77. ENDSUB

78.

### K-Bucket Split

#### What is it?



*Above: “Bucket Splitting” – The Kademlia Protocol Explained Succinctly*

One thing which is not included in the above diagram, is the checking of the “depth” of the K-bucket. Depth is the “longest shared prefix” of ID bits between all contact ID’s in the K-bucket. For example, if the contact IDs of a K-bucket were:

1. 0101111
2. 0101101
3. 0100110
4. 0101011

The depth of the K-bucket would be three, because “010” is the longest prefix of all the IDs, which is three characters long.

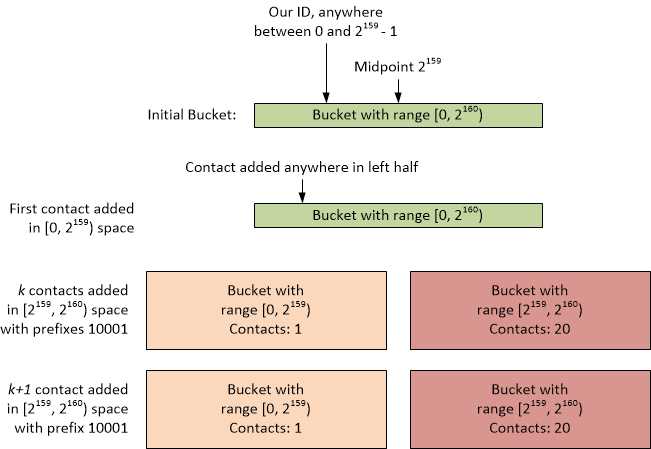
This is used in the specification for checking if a bucket can be split. It does this by checking if **one** of the following is true:

1. Our node ID is close to the contact’s node ID – this is done by making sure that they are in the same K-buckets range.
2. The depth of the K-bucket MOD B (typically 5) does not equal 0. An interview between Brian Muller and Marc Clifton said the following:

“Is the purpose of the depth to limit the number of ‘new’ nodes that a host will maintain (ignoring for the moment the issue of pinging an old contact to see if can be replaced)?” [Brian Muller] replied: “Yes! The idea is that a node should know about nodes spread across the network—though definitely not all of them. The depth is used as a way to control how ‘deep’ a node’s understanding of the network is (and the number of nodes it knows about).”

Some may think that this will cause an issue, because if the K-bucket can’t split, some new contacts may not be able to be added to the peer that they know. But this is handled in due course, as they will be added to a list of pending contacts – as part of a different algorithm, which is not covered here, and they will get a list of “nearby” peers anyway from the known peer either way, and the bootstrapping process can be continued from there. When one of these nearby peers succeeds, the contact ID will be spread through the network.

Splits are handled along powers of 2. For example, in an initial K-bucket (with range 0 < ID < 2160) if there is one node with ID < 2159, and 20 nodes are added with IDs > 2159, The bucket will be split into 2 K-buckets; the first with size 1, and the second with size 20. Marc Clifton made a great diagram of this.



*Above: Degrading adding contacts.*

#### Pseudocode

1. SUB split(self):

2. LET midpoint = (self.low + self.high) INTDIV 2

3. LET k1 = NEW KBucket(low=self.low, high=midpoint)

4. LET k2 = NEW KBucket(low=midpoint, high=self.high)

5.

6. FOR c IN self.contacts:

7. IF c.id.value < midpoint THEN

8. k1.add\_contact(c)

9. ELSE

10. k2.add\_contact(c)

11. ENDIF

12. ENDFOR

13.

14. RETURN k1, k2

15. ENDSUB

16.

#### Notes on pseudocode:

This implementation makes the K-bucket range inclusive on the low end, but exclusive on the high end. I am unsure as to what is proper in Kademlia itself, but this does not seem problematic to me.

### Bootstrap

The general idea of bootstrapping into the network is that “a node ‘u’ must have a contact to an already participating node ‘w’. ‘u’ inserts ‘w’ into the appropriate k-bucket. ‘u’ then performs a node lookup for its own node ID. Finally, ‘u’ refreshes all k-buckets further away than its closest neighbour. During the refreshes, ‘u’ both populates its own k-buckets and inserts itself into the other nodes’ k-buckets as necessary.”

That is what was given from the original specification, but it is incredibly vague.

We can choose a random ID inside node *w’*s contacts bucket range, and then get at most K close contacts to that ID. Distance is calculated by XORing the two IDs together, which is the defining characteristic of Kademlia.

#### Pseudocode

1. SUB random\_id\_within\_bucket\_range(bucket):

2. LET low\_range = bucket.low()

3. LET high\_range = bucket.high()

4. LET random\_offset = random.randint(0, high\_range - low\_range)

5. LET random\_id = low\_range + random\_offset

6. RETURN random\_id

7. END SUB

8.

1. SUB bootstrap(known\_peer):

2. OUR\_BUCKET\_LIST.add\_contact(known\_peer)

3. ADD known\_peer TO self.bucket\_list

4. Contacts = known\_peer.find\_node(key=OUR\_ID)

5. FOR contact IN Contacts

6. ADD contact TO self.bucket\_list

7. ENDFOR

8. Known\_peers\_bucket = self.bucket\_list.get\_kbucket(known\_peer.id)

9. Other\_buckets = buckets IN self.bucketlist EXCLUDING known\_peers\_bucket

10. FOR other\_bucket IN other\_buckets

11. REFRESH other\_bucket

12. ENDFOR

13. ENDSUB

14.

## Networking

### Request Handling

I think an HTTPServer would serve well here. I am not entirely familiar with them, but it is a common python networking module, capable of TCP communication. Due to how commonly its used, it should be relatively easy to find and follow the library’s documentation.

### Request Sending

Since I am using an HTTPServer, I can use the ‘requests’ library to send requests to the server, which implements many HTTP commands.

## Handling global variables

CustomTkinter relies on large scope variables, this can be easily handled using classes, as explained in its documentation. By this, I am referring to calling buttons “self.my\_button”, so that they can then be called in other methods.

## Pseudocode

### ID

“inititalise” will be incorporated into a “\_\_init\_\_” method when I create it in Python.

1. SUB initialise(self, value)

2. Self.MAX\_ID = 2 \*\* Constants.ID\_LENGTH\_BITS

3. SELF.MIN\_ID = 0

4. IF NOT (self.MIN\_ID =< value < self.MAX\_ID)

5. RAISE ValueError(“ID is out of range.”)

6. ENDIF

7. Self.value = value

8. ENDSUB

9.

10. SUB hex(self)

11. RETURN INT\_TO\_HEX(self.value)

12. ENDSUB

13.

14. SUB denary(self)

15. RETURN self.value

16. ENDSUB

17.

18. SUB bin(self)

19. # Converts the denary value into a padded binary number.

20. Binary = INT\_TO\_BIN(self.value)

21. Number\_of\_zeroes\_to\_add = CEIL(LOG(self.MAX\_ID, base=2)) – LENGTH(Binary)

22. Padded\_binary = number\_of\_zeroes\_to\_add \* “0” + binary

23. RETURN padded\_binary

24. ENDSUB

25.

26. SUB big\_endian\_bytes(self)

27. RETURN [x for x in self.bin()]

28. ENDSUB

29.

30. SUB little\_endian\_bytes(self)

31. RETURN self.big\_endian\_bytes().REVERSE()

32. ENDSUB

33.

34.

The following are pseudocode for custom Python “magic methods”, which are called when performing a certain operation (e.g. XOR, addition, subtraction) with the given object.

1. SUB \_\_xor\_\_(self, val)

2. IF isinstance(val, ID)

3. RETURN self.value ^ val.value

4. ENDIF

5. ELSE

6. RETURN self.value ^ val

7. ENDELSE

8. ENDSUB

9.

10. SUB \_\_eq\_\_(self, val)

11. IF isinstance(val, ID)

12. RETURN self.value == val.value

13. ENDIF

14. ELSE

15. RETURN self.value == val

16. ENDELSE

17. ENDSUB

18.

19. SUB \_\_ge\_\_(self, val)

20. IF isinstance(val, ID)

21. RETURN self.value >= val.value

22. ENDIF

23. ELSE

24. RETURN self.value >= val

25. ENDELSE

26. ENDSUB

27.

28. SUB \_\_le\_\_(self, val)

29. IF isinstance(val, ID)

30. RETURN self.value <= val.value

31. ENDIF

32. ELSE

33. RETURN self.value <= val

34. ENDELSE

35. ENDSUB

36.

37. SUB \_\_lt\_\_(self, val)

38. IF isinstance(val, ID)

39. RETURN self.value < val.value

40. ENDIF

41. ELSE

42. RETURN self.value < val

43. ENDELSE

44. ENDSUB

45.

46. SUB \_\_gt\_\_(self, val)

47. IF isinstance(val, ID)

48. RETURN self.value > val.value

49. ENDIF

50. ELSE

51. RETURN self.value > val

52. ENDELSE

53. ENDSUB

54.

55. SUB \_\_str\_\_(self)

56. RETURN INT\_TO\_STR(self.value)

57. ENDSUB

58.

59. SUB \_\_repr\_\_(self)

60. RETURN INT\_TO\_STR(self.value)

61. ENDSUB

62.

### Contact

1. SUB initialise(self, id, protocol)

2. Self.protocol = protocol

3. Self.id = id

4. Self.last\_seen = GET\_CURRENT\_TIME()

5. ENDSUB

6.

7. SUB touch(self)

8. Self.last\_seen = GET\_CURRENT\_TIME()

9. ENDSUB

10.

### KBucket

1. SUB initialise(self, initial\_contacts = list(), low = 0, high = 2^160)

2. Self.contacts = initial\_contacts

3. Self.low: PROTECTED = low

4. Self.high: PROTECTED = high

5. Self.time\_stamp = GET\_CURRENT\_TIME()

6. ENDSUB

7.

8. SUB is\_full(self, other\_id)

9. RETURN LENGTH(self.contacts) >= Constants.K

10. ENDSUB

11.

12. SUB is\_in\_range(self, other\_id)

13. RETURN self.low <= other\_id.value <= self.high

14. ENDSUB

15.

16. SUB add\_contact(self, contact)

17. IF self.is\_full()

18. RAISE TooManyContactsError(“KBucket is full.”)

19. ELSE IF NOT self.is\_in\_range(contact.id)

20. RAISE OutOfRangeError(“Contact ID is out of range.”)

21. ELIF contact not in self.contacts

22. Self.contacts.append(contact)

23. ELSE

24. OUTPUT “Contact already in KBucket.”

25. ENDIF

26. ENDSUB

27.

28. SUB shared\_bits(self)

29. Shared\_bits = GET\_COMMON\_PREFIX([i.id.bin() FOR I IN self.contacts])

30. RETURN shared\_bits

31. ENDSUB

32.

33. SUB depth(self)

34. RETURN LENGTH(self.shared\_bits())

35. ENDSUB

36.

37. The depth is the length of the prefix shared by all nodes binary ID in the k-buckets range. This works because the binary ID is padded with zeroes.

38.

39. SUB split(self)

40. Midpoint = (self.low + self.high) // 2

41. K1 = KBucket(low=self.low, high=midpoint)

42. K2 = KBucket(low=midpoint, high=self.high)

43. FOR c IN self.contacts

44. IF c.id.value < midpoint

45. K1.add\_contact(c)

46. ELSE

47. K2.add\_contact(c)

48. ENDIF

49. ENDFOR

50. RETURN K1, K2

51. ENDSUB

52.

Midpoint will always be an integer because self.low and self.high will be powers of 2. Also this doesn’t put half of the contacts in 1 bucket, and half in the other, because that would not guarantee that the high and low parameters of the created KBucket are powers of 2.

1. SUB replace\_contact(self, contact)

2. Contact\_ids = [c.id FOR c IN self.contacts]

3. Index = contact\_ids.index(contact.id)

4. contact.touch()

5. Self.contacts[index] = contact

6. ENDSUB

7.

8. SUB evict\_contact(self, contact)

9. IF self.contains(contact.id)

10. Self.contacts.remove(contact)

11. ELSE

12. RAISE BucketDoesNotContainContactToEvictError(“Contact not found.”)

13. ENDIF

14. ENDSUB

15.

### Bucket List

1. SUB initialise(self, our\_contact)

2. Self.DHT = None

3. Self.buckets = [KBucket()]

4. Self.our\_id = our\_contact.id

5. Self.our\_contact = our\_contact

6. ENDSUB

7.

The DHT object will be added when the DHT object is created, it will overwrite the default “None” value for self.DHT.

1. SUB can\_split(self, kbucket)

2. RETURN (kbucket.is\_in\_range(self.our\_id) OR kbucket.depth() MOD Constants.B != 0)

3. ENDSUB

4.

The has\_in\_range() argument is testing whether our node ID is close to the contact’s node ID. If our node ID is in the range of the bucket associated with the contact’s node ID, then we know the two nodes are relatively close. Initially, the bucket range spans the entire ID space, so everyone is relatively close. This is refined as more contacts are added.

The “kbucket.depth() MOD Constants.B != 0” argument is performed in order to limit a nodes understanding of a network (or how “deep a node’s understanding is” – Brian Muller), and that it doesn’t experience too much strain from handling requests from new nodes.

1. SUB add\_contact(self, contact)

2. IF self.our\_id == contact.ID

3. RAISE OurNodeCannotBeAContactError(“Cannot add ourselves as a contact.”)

4. ENDIF

5. Touch contact

6. Kbucket = get kbucket for contact.id

7. IF Kbucket contains contact.id

8. Kbucket.replace\_contact(contact)

9. ELIF kbucket.is\_full()

10. IF self.can\_split(kbucket)

11. K1, K2 = kbucket.split()

12. Index = self.get\_kbucket\_index(contact.id)

13. Self.buckets[index] = K1

14. Self.buckets.insert(index + 1, K2)

15. Self.add\_contact(contact)

16. ELSE

17. Last seen contact = SORT\_BY\_LASTSEEN(kbucket.contacts).First()

18. Error = PING last seen contact

19. IF error

20. OUTPUT “Node is unresponsive”

21. DHT delay eviction

22. ELSE

23. OUTPUT “Node is responsive”

24. IF self.DHT

25. Add contact to DHT pending

26. ENDIF

27. ENDIF

28. ENDIF

29. ELSE

30. add contact to kbucket ENDIF

31. ENDSUB

32.

This adds a contact to a k-bucket in the list, this is determined by the range of k-buckets in the lists. This range should span the entire ID space - so there should always be a k-bucket to be added. If we can't add it, it will be added to DHT pending contacts. This raises an error if we try to add ourselves to the k-bucket.

1. SUB get\_close\_contacts(self, key, exclude)

2. Contacts = []

3. FOR bucket IN self.buckets

4. FOR contact IN bucket.contacts

5. IF contact.id != exclude

6. Append contact to contacts

7. ENDIF

8. ENDFOR

9. ENDFOR

10. contacts = sort\_by\_distance(contacts).take(constants.K)

11. RETURN contacts

12. ENDSUB

13.

This performs a ‘brute force’ distance lookup of all known contact then returns the K closest.

### Node

1. SUB initialise(self, contact, storage, cache\_storage)

2. Self.our\_contact = contact

3. Self.storage = storage

4. Self.cache\_storage = cache\_storage

5. Self.DHT = None

6. Self.bucket\_list = BucketList(contact)

7. ENDSUB

8.

9. SUB ping(self, contact)

10. IF sender.id == self.our\_contact.id

11. RAISE Sending query to self error

12. ENDIF

13. Send key values if new contact

14. Add contact to bucket list

15. RETURN self.our\_contact

16. ENDSUB

17.

When this is called, someone is pinging us. We register the contact and respond with our contact.

1. SUB store(self, key, sender, val, is\_cached=False, expiration\_time\_sec)

2. IF sender.id == self.our\_contact.id

3. RAISE Sending query to self error

4. ENDIF

5. Add contact to bucket list

6. IF is\_cached

7. Set key value pair in self.cache\_storage

8. ELSE

9. Send key values if new contact

10. Set key value pair in self.storage

11. ENDIF

12. ENDSUB

13.

This is quite straightforward, and stores a key-value pair in the republish or cache storage.

1. SUB find\_node(self, key, sender)

2. IFsender.id == self.our\_contact.id

3. RAISE Sending query to self error

4. ENDIF

5. Send key values if new contact

6. Add sender to kbucket

7. Contacts = self.bucket\_list.get\_close\_contacts(key, sender.id)

8. RETURN Contacts, None

9. ENDSUB

10.

1. SUB find\_value(self, key, sender)

2. IF sender.id == self.our\_contact.id

3. RAISE sending query to self error

4. ENDIF

5. Send key values if new contact

6. IF key IN self.storage

7. RETURN None, self.storage.get(key)

8. ELIF key IN self.cache\_storage

9. RETURN None, self.cache\_storage.get(key)

10. ELSE

11. RETURN self.bucket\_list.get\_close\_contacts(key, sender.id), None

12. ENDIF

13. ENDSUB

14.

This checks if we have the value, if we don’t, we return K close contacts from the bucket list – these are nodes who are more likely to have it, because it’s closer to the key. This function is called when we receive a FIND\_VALUE RPC, so the results of this will be returned to the sender.

1. SUB send\_key\_values\_if\_new\_contact(self, sender)

2. IF sender is a new contact

3. Contacts = all contacts in our bucket list

4. IF LENGTH(contacts) > 0:

5. FOR k IN self.storage.get\_keys()

6. Distance = MIN([c.id XOR k for c in contacts])

7. IF (self.our\_contact.id XOR k) < distance

8. Sender.protocol.store(k, self.storage.get(k))

9. ENDIF

10. ENDFOR

11. ENDIF

12. ENDIF

13. ENDSUB

14.

If the contact is new, we will tell it to store values where the key XOR our contact is less than the key XOR other contacts.

#### Server entry points

The best to place these are inside Node, so that they can utilise the PING, STORE, FIND\_VALUE and FIND\_NODE methods. These are quite simple, they mostly format the request and use the corresponding method, such as ping. Kademlia asks for random IDs to be sent along with requests to prevent spam attacks.

1. SUB server\_ping(self, request)

2. Self.ping(formatted request data)

3. RETURN {“random\_id”: request[“random\_id”]}

4. ENDSUB

5.

1. SUB server\_store(self, request)

2. Self.store(formatted request data)

3. RETURN {“random\_id”: request[“random\_id”]}

4. ENDSUB

5.

1. SUB server\_find\_node(self, request)

2. Contacts, val = self.find\_node(formatted request data)

3. Contact\_dict = list()

4. FOR c IN Contacts

5. Contact\_info = {“contact”: c.id.value, “protocol”: c.protocol, “protocol\_name”: type(c.protocol)}

6. Contact\_dict.append(contact\_info)

7. ENDFOR

8. RETURN {“contacts”: contact\_dict, “random\_id”: request[“random\_id”]}

9. ENDSUB

10.

1. SUB server\_find\_value(self, request)

2. Contacts, val = self.find\_value(formatted request data)

3. Contact\_dict = list()

4. FOR c IN Contacts

5. Contact\_info = {“contact”: c.id.value, “protocol”: c.protocol, “protocol\_name”: type(c.protocol)}

6. Contact\_dict.append(contact\_info)

7. ENDFOR

8. RETURN {“contacts”: contact\_dict, “random\_id”: request[“random\_id”], “value”: val}

9. ENDSUB

10.

### Routers

#### Base Router

1. SUB initialise(self, node)

2. Self.closer\_contacts = [ ]

3. Self.further\_contacts = [ ]

4. Self.node = node

5. Self.dht = None

6. ENDSUB

7.

1. SUB query(self, key, nodes\_to\_query, rpc\_call, closer\_contacts, further\_contacts)

2. Found = False

3. Found\_by = None

4. Val = “”

5.

6. FOR n IN nodes\_to\_query

7. Found, val, Found\_by, closer\_contacts, further\_contacts = self.get\_closer\_nodes(key, n, rpc\_call, closer\_contacts, further\_contacts)

8. IF found:

9. BREAK

10. ENDFOR

11.

12. RETURN {“found”: found, “contacts”: closer\_contacts, “found\_by”: found\_by, “val”: val}

13. ENDSUB

14.

#### Router (Inherits Base Router)

1. SUB initialise(self, node)

2. BaseRouter.initialise(self, node)

3. ENDSUB

4.

1. SUB lookup(self, key, rpc\_call, give\_me\_all=False)

2. Contacted nodes = [ ]

3. IF TRY CLOSEST BUCKET

4. all\_nodes = get close contacts(k, our id).take(K)

5. nodes\_to\_query = all\_nodes.take(A)

6. Append all all\_nodes to self.further\_contacts

7. ELSE

8. all\_nodes = get close contacts(key, our id)

9. nodes\_to\_query = all\_nodes.take(A)

10.

11. FOR n IN nodes\_to\_query

12. IF n.ID ^ key < (our id ^ key)

13. Append n to self.closer\_contacts

14. ELSE

15. Append n to self.further\_contacts

16. ENDFOR

17. FOR n IN all\_nodes IF n NOT IN nodes\_to\_query

18. Append n to self.further\_contacts

19. ENDFOR

20. ENDIF

21. FOR n IN nodes\_to\_query

22. IF n.id NOT IN [i.id FOR i IN contacted\_nodes]

23. Append n to contacted\_nodes

24. ENDIF

25. ENDFOR

26.

27. Query\_result = self.query(key, nodes\_to\_query, rpc\_call, self.closer\_contacts, self.further\_contacts)

28. IF query\_result[“found”]

29. RETURN query\_result

30. ENDIF

31.

32. ret = [ ]

33. FOR c IN self.closer\_contacts

34. IF c.id NOT IN [i.id FOR i IN ret]

35. Append c to ret

36. ENDIF

37. ENDFOR

38.

39. Have\_work = True

40. WHILE LENGTH(ret) < K and have\_work

41. Closer\_uncontacted\_nodes = self.closer\_contacts EXCLUDING contacted\_nodes

42. Further\_uncontacted\_nodes = self.further\_contacts EXCLUDING contacted\_nodes

43. Have\_closer = LENGTH(closer\_uncontacted\_nodes) > 0

44. Have\_further = LENGTH(further\_uncontacted\_nodes) > 0

45. Have\_work = have\_closer or have\_further

46. IF have\_closer

47. New\_nodes\_to\_query = closer\_uncontacted\_nodes.take(A)

48. FOR c IN new\_nodes\_to\_query

49. IF c.id NOT IN [i.id FOR i IN contacted\_nodes]

50. Append c to Contacted\_notes

51. ENDIF

52. ENDFOR

53. Query\_result = self.query(key, new\_nodes\_to\_query, rpc\_call, self.closer\_contacts, self.further\_contacts)

54. IF query\_result[“found”]

55. RETURN query\_result

56. ENDIF

57. ELSE IF have\_further

58. New\_nodes\_to\_query = further\_uncontacted\_nodes.take(A)

59. FOR c IN new\_nodes\_to\_query

60. IF c.id NOT IN [i.id FOR i IN contacted\_nodes]

61. Append c to Contacted\_notes

62. ENDIF

63. ENDFOR

64. Query\_result = self.query(key, new\_nodes\_to\_query, rpc\_call, self.closer\_contacts, self.further\_contacts)

65. IF query\_result[“found”]

66. RETURN query\_result

67. ENDIF

68. ENDIF

69. ENDWHILE

70. RETURN {“found”: False,

71. “contacts”: (ret IF give\_me\_all else SORT\_BY\_DISTANCE(ret).take(K)),

72. “found\_by”: None,

73. “val”: None}

74. ENDSUB

75.

#### ParallelRouter (Inherits Base Router)

1. SUB initialise(self, node)

2. Super().initialise(node)

3. Self.contact\_queue = InfiniteLinearQueue()

4. Self.semaphore = Semaphore()

5. Self.now = GET\_CURRENT\_TIME()

6. Self.stop\_work = False

7. Initialise thread pool

8. ENDSUB

9.

1. SUB queue\_work(self, key, contact, rpc\_call, closer\_contacts, further\_contacts, find\_result)

2. Self.contact\_queue.enqueue({“key”: key, “contact”: contact, “rpc\_call”: rpc\_call, “closer\_contacts”: closer\_contacts, “further\_contacts”: further\_contacts, “find\_result”: find\_result})

3. Self.semaphore.release()

4. ENDSUB

5.

1. SUB rpc\_caller(self)

2. Flag = True

3. WHILE flag:

4. Self.semaphore.acquire()

5. Item = self.contact\_queue.dequeue()

6. IF item:

7. Found, val, found\_by, item[“closer\_contacts”], item[“further\_contacts”] = self.get\_closer\_nodes(item["key"], item["contact"], item["rpc\_call"], item["closer\_contacts"] , item["further\_contacts"])

8.

9. IF val OR found\_by AND NOT self.stop\_work

10. item["find\_result"]["found"] = True

11. item["find\_result"]["found\_by"] = found\_by

12. item["find\_result"]["found\_value"] = val

13. item["find\_result"]["found\_contacts"] = item["closer\_contacts"]

14. ENDIF

15. ENDIF

16. ENDWHILE

17. ENDSUB

18.

The lookup method employs the same techniques as “Router” lookup, but with more parallelism, I feel like it does not need to be entirely rewritten in pseudocode to convey this.

### DHT

This is the main entry point for our peer to interact with other peers.

1. SUB initialise(self, id, protocol, router, storage\_factory, originator\_storage, republish\_storage, cache\_storage)

2. IF originator\_storage

3. Self.originator\_storage = originator\_storage

4. ELSE

5. Self.originator\_storage = storage\_factory()

6. ENDIF

7. IF republish\_storage

8. Self.republish\_storage = republish\_storage

9. ELSE

10. Self.republish\_storage = storage\_factory()

11. ENDIF

12. IF cache\_storage

13. Self.cache\_storage = cache\_storage

14. ELSE

15. Self.cache\_storage = storage\_factory()

16. ENDIF

17.

18. Self.pending\_contacts = empty list

19. Self.our\_id = id

20. Self.our\_contact = Contact(id, protocol)

21. Self.Node = router.node

22. Self.node.dht = self

23. Self.node.bucket\_list = self

24. Self.protocol = protocol

25. Self.router = router

26. Self.router.node = self.node

27. Self.router.dht = self

28. Self.eviction\_count = {}

29. ENDSUB

30.

1. SUB store(key, val)

2. Touch bucket with key (key)

3. Self.originator\_storage.set(key, val)

4. Store on closer contacts (key, val)

5. ENDSUB

6.

1. SUB find\_value(key)

2. Touch bucket with key (key)

3. Contacts = None

4. Val = None

5. Found, our\_val = try get value from originator storage (key)

6. IF our\_val

7. Found = TRUE

8. Val = our\_val

9. ELSE

10. Found, our\_val = try get value from republish storage (key)

11. IF our\_val

12. Found = TRUE

13. Val = our\_val

14. ELSE

15. Found, our\_val = try get value from cache storage (key)

16. IF our\_val

17. Found = TRUE

18. Val = our\_val

19. ELSE

20. lookup = self.router.lookup(key, self.router.rpc\_find\_value)

21. IF lookup[“found”]

22. Found = True

23. Contacts = None

24. Val = lookup[“val”]

25. Store\_to = First contact in lookup[“contacts”] that != lookup[“found\_by”]

26. IF store\_to

27. Separating\_nodes = get separating nodes count between self.our\_contact and store\_to

28. Exp\_time\_sec = Constants.EXPIRATION\_TIME\_SEC INTDIV (2\*\*separating\_nodes)

29. Store\_to.protocol.store(key, lookup[“val”])

30. ENDIF

31. ENDIF

32. ENDIF

33. ENDIF

34. ENDIF

35. ENDSUB

36.

1. SUB store\_on\_closer\_contacts(self, key, val)

2. Now = GET\_CURRENT\_TIME()

3. Kbucket = get kbucket(key)

4. IF now – kbucket timestamp < BUCKET REFRESH INTERVAL

5. Contacts = get close contacts(key, our id)

6. ELSE

7. Contacts = lookup(key, our id)

8. ENDIF

9. FOR c IN contacts

10. Store(our contact, key, val)

11. ENDFOR

12. ENDSUB

13.

1. SUB bootstrap(self, known\_peer)

2. Add known peer to our bucket list

3. Contacts, error = known\_peer.find\_node(our contact, our id)

4. Handle error

5. IF NOT error

6. FOR contact IN contacts

7. Add contact to our bucket list

8. ENDFOR

9. Known peers bucket = self.node.bucket\_list.get\_kbucket(known\_peer.id)

10.

11. Other\_buckets = all buckets in self.node.bucket\_list that are not known peers bucket

12. Refresh all other buckets

13. ELSE

14. RAISE error

15. ENDIF

16. ENDSUB

17.

1. SUB refresh\_bucket(bucket)

2. Bucket.touch()

3. Random\_id = random id within bucket range

4. Contacts = all contacts in bucket

5. FOR contact in contacts

6. New contacts, timeout error = contact.protocol.find\_node(our contact, random\_id)

7. Handle timeout error

8. Add all new contacts to our bucket list

9. ENDFOR

10. ENDSUB

11.

1. SUB handle\_error(error, contact)

2. IF error.has\_error()

3. Count = add contact to evict()

4. IF count >= EVICTION LIMIT

5. Replace contact

6. ENDIF

7. ENDIF

8. ENDSUB

9.

This increments the number of attempts we have tried to evict the contact that caused the error, after EVICTION LIMIT tries we will replace the contact.

1. SUB delay\_eviction(self, to\_evict, to\_replace)

2. IF to\_replace NOT IN pending contacts

3. APPEND to\_replace to pending contacts

4. ENDIF

5. Key = to\_evict.id.value

6. Count = add contact to evict (key)

7. IF count == EVICTION LIMIT

8. Replace\_contact(to\_evict)

9. ENDIF

10. ENDSUB

11.

1. SUB save(self, filename)

2. OPEN(filename) as output\_file

3. PICKLE.dump(self) to output\_file

4. CLOSE output\_file

5. ENDSUB

6.

1. SUB load(filename)

2. OPEN(filename) as input\_file

3. Data = PICKLE.load(input\_file)

4. CLOSE input\_file

5. RETURN data

6. ENDSUB

7.

### Networking

#### Base Server

1. SUB initialise(self, server\_address, RequestHandlerClass)

2. HTTPServer.initialise(self, server\_address, RequestHandlerClass)

3. self.routing\_methods= {

4. "/ping": PingRequest,

5. "/store": StoreRequest,

6. "/find\_node": FindNodeRequest,

7. "/find\_value": FindValueRequest

8. }

9. ENDSUB

10.

#### HTTP Request Handler

1. SUB common\_request\_handler(self, method\_name, common\_request, node)

2. Old\_self = self // Prevents other threads from overwriting it

3.

4. TRY

5. Method = get\_method(node, method\_name)

6. Response = method(common\_request)

7. Encoded\_response = encode\_data(response)

8. Send response (code=200)

9. Send body (encoded\_response)

10. CATCH Exception as e

11. Error\_response = {“error\_message”: e.message, “random\_id” = RANDOM\_ID()}

12. Encoded\_response = encode\_data(error\_response)

13. Send response (code=400)

14. Send body (encoded response)

15. ENDTRYCATCH

16. ENDSUB

17.

1. SUB handle\_post(self)

2. routing\_methods = {

3. "/ping": PingRequest,

4. "/store": StoreRequest,

5. "/find\_node": FindNodeRequest,

6. "/find\_value": FindValueRequest

7. }

8. Encoded\_request = self.request.read()

9. Decoded\_request = decode\_data(encoded\_request)

10. Request\_dict = decoded\_request

11. Path = self.path

12. Method\_name = “server\_” + path.SUBSTRING(1, -1)

13.

14. Request\_type = routing\_methods[path]

15.

16. IF request\_type != None

17. Common\_request = {

18. “protocol”: request\_dict[“protocol”],

19. “protocol\_name”: request\_dict["protocol\_name”],

20. “random\_id”: request\_dict["random\_id”],

21. “sender”: request\_dict["sender”],

22. “key”: request\_dict["key”],

23. “value”: request\_dict["value”],

24. “is\_cached”: request\_dict[“is\_cached"],

25. “expiration\_time\_sec”: request\_dict[“expiration\_time\_sec”]

26. }

27. Node = self.server.node

28. IF node

29. Self.common\_request\_handler(method\_name, common\_request, node)

30. ELSE

31. Encoded\_response = encode\_data({“error\_message”: “Node not found”})

32. Send response (code=400)

33. Send body (encoded\_response)

34. ENDIF

35. ENDIF

36. ENDSUB

37.

### GUI

#### Contact Viewer (Inherits CustomTkinter main window)

1. SUB initialise(id, protocol\_type, url, port)

2. initialise(CustomTkinter main window)

3. self.SET\_TITLE(“Contact Viewer”)

4. Self.id = id

5. Self.protocol\_type = protocol\_type

6. Self.url = url

7. Self.port = port

8. MAKE Label(text = “Our contact:”)

9. MAKE Label(text = “ID: ” + INT\_TO\_STR(self.id))

10. MAKE Label(text = “Protocol type: “ + TYPE\_TO\_STR(self.protocol\_type))

11. MAKE Label(text = “URL: “ + self.url)

12. MAKE Label(text = “Port” + INT\_TO\_STR(self.port))

13. MAKE Button(text=”Export our contact”, command=”self.export\_contact”)

14. ENDSUB

15.

1. SUB export\_contact(self, filename = “our\_contact.json”)

2. contact\_dict = {

3. "url": self.url,

4. "port": self.port,

5. "protocol\_type": str(self.protocol\_type),

6. "id": self.id

7. }

8. OPEN filename AS f

9. DUMP\_JSON(contact\_dict, f)

10. CLOSE f

11. SHOW STATUS WINDOW “Exported our contact to { }”.format(filename)

12. ENDSUB

13.

#### Settings (Inherits CustomTkinter main window)

1. SUB initialise(self, hash\_table)

2. initialise(CustomTkinter main window)

3. self.dht = hash\_table

4. self.SET\_TITLE(“Kademlia Settings”)

5. MAKE Label(text=”Settings”)

6. MAKE Label(text=”File to export to: “)

7. Self.export\_file\_textbox = Textbox()

8. MAKE self.export\_file\_textbox

9. MAKE Button(text=”Export/Save DHT”, command=self.export\_dht)

10. MAKE Button(text=”View our contact”, command=self.view\_contact)

11. ENDSUB

12.

1. SUB export\_dht(self)

2. File = self.export\_file\_textbox.READ\_TEXTBOX()

3. TRY

4. Self.dht.save()

5. SHOW STATUS WINDOW “File saved successfully.”

6. CATCH Exception as e

7. SHOW ERROR WINDOW e.message

8. ENDTRYCATCH

9. ENDSUB

10.

1. SUB view\_contact(self)

2. Our\_contact = self.dht.our\_contact

3. Our\_id = our\_contact.id.value

4. Protocol = our\_contact.protocol

5. Protocol\_type = GET\_TYPE(protocol)

6. Our\_ip\_address = protocol.url

7. Our\_port = protocol.port

8.

9. RUN ContactViewer(

10. id=our\_id,

11. protocol\_type=protocol\_type,

12. url=our\_ip\_address,

13. port=our\_port,

14. appearance\_mode=self.appearance\_mode

15. )

16. ENDSUB

17.

#### Main GUI (Inherits from CustomTkinter main window)

1. SUB inititalise(self)

2. Initalise(CustomTkinter main window)

3. Self.SET\_TITLE(“Kademlia”)

4. Make\_dht\_join\_frame()

5. ENDSUB

6.

1. SUB initialise\_kademlia(self)

2. Our\_id = RANDOM\_ID()

3. Our\_ip = get\_ip()

4. Valid\_port = get\_random\_port()

5. Protocol = TCPProtocol(our\_ip, valid\_port)

6. Our\_contact = Contact(our\_id, protocol)

7. Our\_node = Node(

8. our\_contact,

9. SecondaryJSONStorage(INT\_TO\_STR(our\_id.value) + “/node.json”),

10. VirtualStorage()

11. Self.dht = DHT(

12. id=our\_id,

13. protocol=protocol,

14. originator\_storage=SecondaryJSONStorage(INT\_TO\_STR(our\_id.value) + “originator\_storage.json"),

15. republish\_storage=SecondaryJSONStorage(INT\_TO\_STR(our\_id.value) + “republish\_storage.json"),

16. cache\_storage=VirtualStorage(),

17. router=ParallelRouter(our\_node)

18. )

19. Self.server = TCPServer(our\_node)

20. Self.server\_thread = Self.server.thread\_start()

21.

22. Make\_network\_frame()

23. ENDSUB

24.

#### Join Network Menu Frame (Inherits CustomTkinter frame)

1. SUB initialise(self, parent)

2. INITIALISE CTkFrame INTO Parent

3. Self.parent = parent

4. MAKE Label(“Join Network”)

5. MAKE Button(“Bootstrap into existing network”, command=make\_load\_dht\_frame)

6. MAKE Button(“Join stored network”, command=make\_bootstrap\_frame)

7. MAKE Button(“Create new network”)

8. ENDSUB

9.

#### Upload Frame (Inherits CustomTkinter Frame)

1. SUB initialise(self, parent)

2. INITIALISE CTkFrame INTO parent

3. Self.parent = parent

4. MAKE Label(“Upload file”)

5. MAKE Label(“File to upload”)

6. Self.Enter\_file\_textbox = Textbox()

7. MAKE Self.enter\_file\_textbox

8. MAKE Button(“Upload”, command=handle\_upload)

9. ENDSUB

10.

1. SUB handle\_upload(self)

2. File\_to\_upload = self.enter\_file\_textbox.GET\_TEXT()

3. IF IS\_VALID\_FILE(File\_to\_upload)

4. Base\_filename = get\_Basename(file\_to\_upload)

5. IF NOT base\_filename

6. SHOW ERROR WINDOW “Must be a file.”

7. ELSE

8. OPEN file\_to\_upload AS f

9. file\_contents = f.read()

10. CLOSE f

11. Val = DUMPS({“filename”: base\_filename, “file”: file\_contents})

12. Id\_to\_store\_to = RANDOM\_ID()

13. STORE(id\_to\_store, val)

14. SHOW STATUS WINDOW “Stored file.”

15.

16. ENDIF

17. ENDSUB

18.

This stores the the file data as a pickled dictionary containing filename, and the filedata. The filename is necessary, so that the recipient can understand what type of file it is, and it supplies additional context.

#### Download Frame (Inherits CustomTkinter Frame)

1. SUB initialise(self, parent)

2. INITIALISE CTkFrame INTO parent

3. Self.parent = parent

4. MAKE Label(“Download file:”)

5. MAKE Label(“ID to download: “)

6. Self.enter\_id\_textbox = Textbox()

7. MAKE self.enter\_id\_textbox

8. MAKE Button(“Download”, command=handle\_download)

9. ENDSUB

10.

1. SUB handle\_download(self)

2. Id\_from\_textbox = self.enter\_id\_textbox.GET\_TEXT()

3. IF NOT id\_from\_textbox:

4. SHOW ERROR WINDOW “ID must not be empty.”

5. ELSE IF NOT id\_from\_textbox.IS\_NUMERIC()

6. SHOW ERROR WINDOW “ID was not a number.”

7. ELSE IF NOT (0 <= STR\_TO\_INT(id\_from\_textbox) < 2Constants.ID\_LENGTH\_BITS)

8. SHOW ERROR WINDOW “ID out of range.”

9. ELSE

10. Id\_to\_download = ID(STR\_TO\_INT(id\_from\_textbox))

11. Found, contacts, val = find\_value(id\_to\_download)

12. IF NOT found

13. SHOW ERROR WINDOW “File not found.”

14. ELSE

15. File\_dict = pickle LOADS(val\_bytes)

16. Filename = file\_dict[“filename”]

17. File\_bytes = file\_dict[“file”]

18. DELETE FROM MEMORY file\_dict

19. Current\_working\_directory = GET\_CWD()

20. OPEN current\_working\_directory + filename AS f

21. f.WRITE\_TO\_FILE(file\_bytes)

22. CLOSE f

23. SHOW STATUS WINDOW “File downloaded.”

24. ENDIF

25. ENDIF

26. ENDSUB

27.

This checks to ensure the ID given is an integer in the range of my implementation, if it is not, the corresponding error window is raised to the user. If the ID is valid, FIND\_VALUE will be called to get the file, and then it will be downloaded its filename in the current working directory.

#### Main Network Frame (Inherits CustomTkinter Frame)

1. SUB initalise(self, parent)

2. INITIALISE CTkFrame INTO parent

3. Self.parent = parent

4. MAKE Label(“Kademlia”)

5. MAKE Button(“Download”, command=make\_download\_frame)

6. MAKE Button(“Upload”, command=make\_upload\_frame)

7. ENDSUB

8.

#### Load DHT from File Frame (Inherits CustomTkinter Frame)

1. SUB initialise(self, parent)

2. INITIALISE CtkFrame INTO parent

3. Self.parent = parent

4. MAKE Label(“Load DHT from file”)

5. MAKE Label(“Load from file: “)

6. Self.filename\_textbox = Textbox()

7. MAKE Button(“Back”, command=make\_join\_dht\_frame)

8. MAKE Button(“Load DHT”, command=load\_dht

9. ENDSUB

10.

1. SUB load\_dht(self)

2. Loaded\_dht = DHT.LOAD(self.filename.textbox.GET\_TEXT)

3. STARTSERVER ON NEW THREAD (self.parent.dht.node)

4. Make\_network\_frame()

5. ENDSUB

6.

#### Join Network Menu Frame (Inherits CustomTkinter Frame)

1. SUB initialise(self, parent)

2. INITIALISE CTkFrame INTO parent

3. Self.parent = parent

4. MAKE Label(“Join Network”)

5. MAKE Button(“Join stored network”, command=make\_load\_dht\_frame)

6. MAKE Button(“Bootstrap into existing network”, command=make\_bootstrap\_frame)

7. MAKE Button(“Create new network”, command=initialise\_kademlia)

8. ENDSUB

9.

#### Bootstrap from JSON (Inherits CustomTkinter Frame)

1. SUB initialise(self, parent)

2. INITIALISE CTkFrame INTO parent

3. Self.parent = parent

4. MAKE Label(“Bootstrap from JSON”)

5. MAKE Label(“Filename: “)

6. Self.filename\_entry = Textbox()

7. MAKE self.filename\_entry

8. MAKE Button(“Back”, command=make\_join\_dht\_frame)

9. MAKE Button(“Load”, command=load\_known\_peer\_json\_for\_bootstrap)

10. ENDSUB

11.

#### Bootstrap (Inherits CustomTkinter Frame)

1. SUB initialise(self, parent)

2. INITIALISE CTkFrame INTO parent

3. Self.parent = parent

4. MAKE Label(“IP address: “)

5. Self.ip\_entry = Textbox()

6. MAKE self.ip\_entry

7. MAKE Label(“Port: “)

8. Self.port\_entry = Textbox()

9. MAKE self.port\_entry

10. MAKE Label(“ID: “)

11. Self.id\_entry = Textbox()

12. MAKE self.id\_entry

13. MAKE Button(“Back”, command=make\_join\_dht\_frame)

14. MAKE Button(“Load from file”, command=make\_bootstrap\_from\_json\_frame)

15. MAKE Button(“Connect”, command = handle\_bootstrap)

16. ENDSUB

17.

1. SUB handle\_bootstrap(self)

2. Valid = False

3. Known\_ip = self.ip\_entry.GET\_TEXT()

4. ip\_regex = "(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])"

5. IF NOT known\_ip

6. SHOW ERROR WINDOW “IP address must not be empty.”

7. ELIF NOT REGEX\_MATCH(known\_ip, ip\_regex)

8. SHOW ERROR WINDOW “IP address is invalid.”

9. ELSE

10. Valid = True

11. ENDIF

12.

13. Known\_port\_str = self.port\_entry.GET\_TEXT()

14. Known\_port = None

15. IF NOT known\_port\_str

16. SHOW ERROR WINDOW “Port must not be empty.”

17. ELIF NOT known\_port\_str.IS\_NUMERIC()

18. SHOW ERROR WINDOW “Port is not numeric.”

19. ELIF NOT (0 < STR\_TO\_INT(known\_port\_str) < 65535)

20. SHOW ERROR WINDOW “Port out of range, must be between 0 and 65535.”

21. ELSE

22. Known\_port =STR\_TO\_INT(known\_port\_str)

23. Valid = True

24. ENDIF

25.

26. IF valid

27. BOOTSTRAP(parent, known\_id, known\_ip, known\_port)

28. ENDIF

29. ENDSUB

30.

1. SUB bootstrap(parent, known\_id, known\_url, known\_port)

2. Known\_protocol = TCPProtocol(known\_url, known\_port)

3. Known\_contact = Contact(known\_id, known\_protocol)

4. Parent.dht.bootstrap

5. ENDSUB

6.

## Class Attributes, Methods and initialisation

Here are a series of tables displaying each class I intend to create in my project, and the methods and values they should contain. I will use the standard Python syntax for public, protected, and private methods/values - no prefix for public, “\_” prefix for protected, “\_\_” prefix for private, and any “magic methods” are given the standard “\_\_function\_\_”, as they are in python, I will not include \_\_init\_\_() in any of these, but it can be assumed for all that it exists.

Here is the table for the Kademlia implementation:

### Buckets.py

#### Kbucket

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| Contacts | public | List[contact] | List of contacts contained inside the KBucket. |
| low | protected | int | Lower bound of IDs allowed in the bucket. |
| high | protected | int | Upper bound of IDs allowed in the bucket. |
| Time\_stamp | public | datetime | Last time when the bucket was edit. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, Initial\_contacts = None, low = 0, high = 2 ^ 160 | None | Initialises Kbucket with a set of initial contacts, and a high-low boundary. |
| Low | public | self | int | Returns self.\_low |
| High | public | self | int | Returns self.\_high |
| Is\_full | public | self | bool | Returns if the kbucket is full. |
| contains | public | Self, id | bool | Returns if the kbucket contains the given ID. |
| touch | public | self | None | Updates time\_stamp to current time. |
| Add\_contact | public | Self, contact | None | Attempts to add contact to bucket. |
| depth | public | self | int | Returns the number of shared bits in the prefix of all IDs in the bucket. |
| Shared\_bits | public | self | str | Returns the shared prefix in bits between all IDs in our bucket. |
| Split | public | self | tuple | Splits the bucket in half along the midpoint of ID range. |
| Replace\_contact | public | Self, contact | None | Replaces a contact in our bucket with another contact with the same ID. |
| Evict\_contact | public | Self, contact | None | Removes a given contact from our bucket list. |

#### BucketList

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| dht | public | DHT | The DHT this bucket list is a part of. |
| buckets | public | List[KBucket] | List of Kbuckets in the bucket list. |
| Our\_id | public | ID | The ID of our bucket lists contact. |
| Our\_contact | public | Contact | The contact that owns this bucket list. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, our\_contact | None | Initialises the bucket list. |
| Can\_split | public | Self, kbucket | bool | Checks if the bucket can split |
| Get\_kbucket\_index | protected | Self, other\_id | int | Returns the first k-bucket index in the bucket list which has a given ID in range. Returns –1 if not found. |
| Get\_kbucket | public | Self, other\_id | KBucket | Returns the first Kbucket in the bucket list which has a given ID in range. |
| Add\_contact | public | Self, contact | None | Adds a contact to a k-bucket in the list, this is determined by the range of k-buckets in the lists. This range should span the entire ID space - so there should always be a k-bucket to be added.  If we can't add it, it will be added to DHT pending contacts.  This raises an error if we try to add ourselves to the k-bucket. |
| Get\_close\_contacts | public | Self, key, exclude | List[Contact] | Brute force distance lookup of all known contacts, sorted by distance. Then we take K of the closest. :param key: The ID for which we want to find close contacts. :param exclude: The ID to exclude (the requesters ID). :return: List of K contacts sorted by distance. |
| contacts | public | self | List[Contact] | Returns a list of all contacts in the bucket list. :return: All contacts in the bucket list. |
| Contact\_exists | public | Self, contact | bool | Returns if a contact is in the bucket list. |

### Constants.py

#### Constants

This is a dataclass containing data which has been discussed at length when I define the constants I will use in my project. It contains no methods, and all attributes are constants that have been previous sections.

### Contacts.py

#### Contact

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| protocol | public | IProtocol | Protocol used by this contact |
| id | public | ID | This contacts ID. |
| Last\_seen | public | datetime | When this contact was last interacted with. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, id, protocol | None | Initialises our contact by setting attributes “self.id” to id, “self.protocol” to protocol, and creating a new attribute “self.last\_seen”. |
| touch | public | self | None | Updates self.last\_seen to the current time. |

### Dht.py

#### DHT

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| Originator\_storage | Protected | IStorage |  |
| Republish\_storage | Protected | IStorage |  |
| Cache\_storage | Protected | IStorage |  |
| Pending\_contacts | Public | List[Contact] | All contacts pending eviction. |
| Our\_id | public | ID | Our ID |
| Our\_contact | public | Contact | Our contact, containing our ID and our protocol |
| node | public | Node | Our node |
| protocol | protected | IProtocol | Our protocol |
| router | protected | BaseRouter | The router used by this DHT |
| Eviction\_count | public | Dict[int, int] | Key value pairs of {id value -> number of times it has been contacted. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, id, protocol, router, storage\_factory, originator\_storage, republish\_storage, cache\_storage | None | Initialises the DHT object, if storage is not provided, then it will be created with storage\_factory. |
| router | public | self | BaseRouter | Returns self.\_router |
| protocol | public | self | IProtocol | Returns self.\_protocol |
| Originator\_storage | public | self | IStorage | Returns self.\_originator\_storage |
| store | public | self, key, val | None | Stores key-value pair in originator storage, touches bucket with given key, and stores the key on closer contacts. |
| Find\_value | public | Self, key | Tuple[bool, list[Contact], str] | ttempts to find a given value. First it checks our originator storage. If the given key does not have a value in our storage, it will use Router.lookup() to attempt to find it. If there is no value found from router.lookup(), the value returned will be None. If there is a value found from router.lookup(), the value will be stored on the closest contact to us, if one exists. :param key: Key to search for value pair. :return: Found: bool (If it is found or not), contacts: list[Contact], val: str | None (value returned) |
| Touch\_bucket\_with\_key | public | self, key | None | Touches a bucket with a given key from the bucket list. |
| Store\_on\_closer\_contacts | public | Self, key, val | None | Stores a value on K close contacts to the key. |
| bootstrap | public | Self, known\_peer | None | This is how we join the network.  We bootstrap our peer by contacting a known peer in the network, adding its contacts to our list, then getting the contacts for other peers not in the bucket range of our known peer we're joining. |
| Refresh\_bucket | protected | Self, bucket | None | Refreshes the given Kademlia KBucket by updating its last-touch timestamp, obtaining a random ID within the bucket's range, and attempting to find nodes in the network with that random ID.  The method touches the bucket to update its last-touch timestamp, generates a random ID within the bucket's range, and queries nodes in the network using the Kademlia protocol to find nodes with the generated ID. If successful, the discovered contacts are added to the Kademlia node's bucket list.  Note: The contacts collection for the given bucket might change during the operation, so it is isolated in a separate list before iterating over it. |
| Setup\_bucket\_refresh\_timer | protected | self | None | Sets up the refresh timer to re-ping buckets. |
| Bucket\_refresh\_timer\_elapsed | protected | Self | None | Refresh all buckets in self.node.bucket\_list |
| Key\_value\_republished\_elapsed | protected | self | None | Replicate key values if the key value hasn't been touched within the republish interval. Also don't do a FindNode lookup if the bucket containing the key has been refresed within the refresh interval. |
| Expire\_keys\_elapsed | protected | Self | None | Removes expired key-values from republish and cache storage. |
| Remove\_expired\_data | protected | store | None | Removes expired data from a given storage object. |
| Originator\_republish\_elapsed | protected | self | None | Redistributes expired key-value pars if we are the publisher. |
| Get\_separating\_nodes\_count | protected | Self, contact\_a, contact\_b | int | Returns the number of contacts between 2 contacts in our bucket list. |
| Handle\_error | public | Self, error, contact | None | Put the timed out contact into a collection and increment the number of times it has timed out. If it has timed out a certain amount, remove it from the bucket and replace it with the most recent pending contact that are queued for that bucket. |
| Delay\_eviction | public | Self, to\_evict, to\_replace | None | The contact that did not respond (or had an error) gets "n" tries before being evicted and replaced with the most recently seen contact that wants to got into the non-responding contact’s bucket.  :param to\_evict: The contact that didn't respond.  :param to\_replace: The contact that can replace the  non-responding contact. |
| Add\_contact\_to\_evict | protected | Self, key | int | Increments how many times we have tried to evict a given key, returning number of attempts. |
| Replace\_contact | protected | Self, to\_evict | None | Replaces an evicted contact with a pending one. |
| Evict\_contact | protected | Self, bucket, to\_evict | None | Removes all attempts to evict to\_evict, then removes it from the given bucket, raising an error if it is not in the bucket. |
| save | public | Self, filename | None | Saves DHT to file. |
| load | public | Cls, filename | DHT | Loads DHT from file. |
| Replace\_with\_pending\_contact | protected | Self, bucket | None | Find a pending contact that goes into the bucket that now has room; that pending contact is no longer pending. |

### Dictionaries.py

#### FindResult(TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| Contacts | List[Contact] | Contacts close to the key in the key-value pair, these are more likely to be able to locate the value we seek. This is empty if the value is found. |
| Val | Str | None | Value from a key-value pair – could be a file etc. |
| Found | Bool | If the value has been found. |
| Found\_by | Contact | None | Who the value was found by. |

#### ContactQueueItem(TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| key | ID | Key of key-value pair |
| contact | Contact | Contact |
| Rpc\_call | Callable | RPC function which will be applied |
| Closer\_contacts | List[Contact] | List of contacts closer to “key” than “contact” |
| Further\_contacts | List[Contact] | List of contacts further than “contact” to “key” |
| Find\_result | FindResult | Result of lookup, that is passed in with this. This has elements ‘value’, ‘contacts’, ‘found\_by’ and ‘found’ |

#### GetCloserNodesReturn(TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| found | bool | If the value was found |
| val | Str | None | Value in key-value pair |
| Found\_by | Contact | None | Who the value was found by – if it was found |

#### BaseRequest(TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| protocol | IProtocol | Protocol used in the request |
| Protocol\_name | str | Name of the protocol used |
| sender | int | Key of sender contact |
| Random\_id | int | Random\_id to prevent spam attacks |

#### FindNodeRequest(BaseRequest, TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| … | … | All from BaseRequest |
| key | int | Key that we are looking for nodes close to. |

#### FindValueRequest(BaseRequest, TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| … | … | All from BaseRequest |
| key | int | Key that corresponds to the value we are looking for in the key-value pair. |

#### PingRequest(BaseRequest, TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| … | … | All from BaseRequest |

#### StoreRequest(BaseRequest, TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| … | … | All from BaseRequest |
| key | int | Key in key-value that we are storing |
| val | str | Value in key-value that we are storing. |
| Is\_cached | bool | If the key-value should be cached instead of stored in republish storage |
| Expiration\_time\_sec | int | How long before the key-value pair expires |

#### ITCPSubnet(TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| subnet | int | Subnet number – this corresponds to a node in TCPSubnetServer / TCPSubnetProtocol, this is used for testing on a single IP-port. |

#### FindNodeSubnetRequest(FindNodeRequest, ITCPSubnet, TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| … | … | All from FindNodeRequest |
| … | … | All from ITCPSubnet |

#### FindValueSubnetRequest(FindValueRequest, ITCPSubnet, TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| … | … | All from FindValueRequest |
| … | … | All from ITCPSubnet |

#### PingSubnetRequest(PingRequest, ITCPSubnet, TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| … | … | All from PingRequest |
| … | … | All from ITCPSubnet |

#### StoreSubnetRequest(StoreRequest, ITCPSubnet, TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| … | … | All from StoreRequest |
| … | … | All from ITCPSubnet |

#### CommonRequest(TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| protocol | IProtocol | Protocol used |
| Protocol\_name | str | Name of protocol used |
| Random\_id | int | Random\_id to prevent spam attacks |
| sender | int | ID of sender as an integer |
| key | int | Key of key-value pair we are looking for |
| value | Str | None | Value of key-value pair |
| Is\_cached | bool | If the key-value should be cached instead of stored in republish storage |
| Expiration\_time\_sec | int | How long before the key-value pair expires |

#### BaseResponse(TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| Random\_id | int | To prevent spam attacks |

#### ErrorResponse(BaseResponse, TypedDict)

|  |  |  |
| --- | --- | --- |
| Element | Type | Description |
| … | … | All from BaseResponse |
| Error\_message | str | Error message from server. |

#### StoreValue(TypedDict)

|  |  |  |
| --- | --- | --- |
| Elements | Type | Description |
| Value | str | Value to store |
| Republish\_timestamp | Str (ISOFormat datetime object) | Time that it was added to storage |
| Expiration\_time | int | Time taken for it to expire |

### Errors.py

#### DataDecodingError (Exception)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from Exception |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| … | … | … | … | Inherited from Exception |

#### TooManyContactsError (Exception)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from Exception |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| … | … | … | … | Inherited from Exception |

#### OutOfRangeError (Exception)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from Exception |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| … | … | … | … | Inherited from Exception |

#### OurNodeCannotBeAContactError (Exception)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from Exception |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| … | … | … | … | Inherited from Exception |

#### AllKBucketsAreEmptyError (Exception)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from Exception |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| … | … | … | … | Inherited from Exception |

#### SendingQueryToSelfError (Exception)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from Exception |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| … | … | … | … | Inherited from Exception |

#### SenderIsSelfError (Exception)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from Exception |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| … | … | … | … | Inherited from Exception |

#### RPCError (Exception)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from Exception |
| Protocol\_error\_message | public | Str | None | Error message, if raised from protocol. |
| Protocol\_error | public | bool | If there is a protocol error. |
| Timeout\_error | public | bool | If there is a timeout error. |
| Id\_mismatch\_error | public | bool | If there is an ID mismatch error. |
| Peer\_error\_message | public | Str | None | Error message from peer, if peer error. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| … | … | … | … | Inherited from Exception |
| \_\_init\_\_ | N/A | self, error\_message, timeout\_error, id\_mismatch\_error, peer\_error, peer\_error\_message | None | Initialises an RPCError method – having all these error types together allows checking for RPCErrors very easy, and still readable. |
| Has\_error | public | self | bool | Returns True if any type of error is true, else False. |
| No\_error | public | cls | RPCError | Returns an empty RPCError. |
| \_\_str\_\_ | N/A | self | str | Returns error message, or “No error” if there is none. |

#### ValueCannotBeNoneError (Exception)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from Exception |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| … | … | … | … | Inherited from Exception |

#### UnknownRequestError(Exception)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from Exception |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| … | … | … | … | Inherited from Exception |

### ID.py

#### ID

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| MAX\_ID | public | Int | Maximum value -exclusive – set to 2ID\_LENGTH\_BITS |
| MIN\_ID | Public | Int | Minimum value - inclusive – set to 0. |
| value | public | int | ID value |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, value | None | Initialises value. |
| hex | public | self | str | Returns value in hex |
| decimal | public | self | int | Returns self.value (decimal) |
| bin | public | self | str | Returns self.value in binary – excluding “0b” tag at the start. |
| Big\_endian\_bytes | public | self | List[str] | Returns self.value in big endian binary as a list. |
| Big\_endian\_bytes | public | self | List[str] | Returns self.value in little endian binary as a list. |
| \_\_xor\_\_ | N/A | Self, val | int | Allows for XOR of ID – ID, or ID – int. Does not allow int – ID as that calls int.\_\_xor\_\_(). |
| \_\_eq\_\_ | N/A | Self, val | int | Allows for equality checking (==) of ID – ID, or ID – int. Does not allow int – ID as that calls int.\_\_eq\_\_(). |
| \_\_ge\_\_ | N/A | Self, val | int | Allows for greater than or equal to comparison of ID – ID, or ID – int. Does not allow int – ID as that calls int.\_\_ge\_\_(). |
| \_\_le\_\_ | N/A | Self, val | int | Allows for less than or equal to comparison of ID – ID, or ID – int. Does not allow int – ID as that calls int.\_\_le\_\_(). |
| \_\_lt\_\_ | N/A | Self, val | int | Allows for less than comparison of ID – ID, or ID – int. Does not allow int – ID as that calls int.\_\_lt\_\_(). |
| \_\_gt\_\_ | N/A | Self, val | int | Allows for greater than comparison of ID – ID, or ID – int. Does not allow int – ID as that calls int.\_\_gt\_\_(). |
| \_\_str\_\_ | N/A | Self | str | Returns self.value as a string. |
| \_\_repr\_\_ | N/A | self | str | Returns self.value. |

### Interfaces.py

#### IStorage

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| contains | public | Self, key | bool | Returns if the given key is contained in the storage object. |
| Try\_get\_value | public | Self, key | Tuple[bool, str] | Attempts to get a value from a key-value pair, returns ‘False, “”’ if it is not there; returns ‘True, value’ if it is there. |
| get | public | Self, key | str | Tries to return value from key-value pair, given key. |
| Get\_timestamp | public | Self, key | datetime | Returns timestamp of key-value pair, given key. |
| set | public | Self, key, value, expiration\_time\_sec | None | Sets key-value pair with expiration time. |
| Get\_expiration\_time\_sec | public | Self, key | int | Gets expiration time from key-value pair, given key. |
| Remove | public | Self, key | None | Removes key-value pair from the storage object, given key. |
| Get\_keys | public | self | List[int] | Returns all keys from key-value pairs stored in the storage object. |
| Touch | public | Self, key | None | Sets timestamp of key-value to current time, given key. |

#### IProtocol

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| node | public | Node | Node that the protocol belongs to. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| ping | public | Self, sender | RPCError | Handles an incoming ping request from “sender”, returns an RPCError object. |
| Find\_node | public | Self, sender, key | Tuple[list[Contact], RPCError] | Attempts to find K close nodes to key, returning them and an RPCError object. |
| Find\_value | public | Self, sender, key | Tuple[list[Contact], str, RPCError] | Attempts to find value from key-value pair, if it cannot be found, a list of K closer contacts are returned. An RPCError object is returned as well to indicate any errors that have occurred. |
| store | public | Self, sender, key, val, is\_cached, exp\_time\_sec | RPCError | Attempts to save a key-value pair to storage, it caches it instead of storing if is\_stored, and it expires after exp\_time\_sec. |

### Locker.py

#### WithLock

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| lock | private | Threading.Lock | Existing lock object. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, lock | None | Creates an object which allows an existing threading.Lock object to be used in “with” statements. |
| \_\_enter\_\_ | N/A | self | None | Locks the lock (this occurs at the start of the ‘with’ statement) |
| \_\_exit\_\_ | N/A | Self, exc\_type, exc\_value, traceback | None | Unlocks the lock (this occurs at the end of the ‘with’ statement) |

### My\_queues.py

#### InfiniteLinearQueue

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| items | private | list | List of items in the queue |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | self | None | Makes a linear queue with no maximum size. |
| Is\_empty | public | self | bool | Returns if the queue is empty or not. |
| enqueue | public | Self, item | None | Adds an item to the queue. |
| dequeue | public | self | Optional[any] | Removes an item to the queue and returns it – returns None if it is not in the queue. |

#### LinearQueue

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| Queue | private | List | The list of items which represent the queue |
| Front | private | Int | The front of the queue, set to 0 |
| Rear | private | Int | The tail of the queue, set to -1 so that the first item is added at index 0 |
| size | private | Int | The size of the queue, set to 0 |
| Max\_size | private | int | Maximum size of the queue. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Descriptions |
| \_\_init\_\_ | N/A | Self, Queue\_size | None | Initialises the queue with a given maximum size. |
| enqueue | public | Self, item | bool | Adds item to the tail of the queue. Returns True if success, False if failure. |
| dequeue | public | self | Optional[any] | Removes an item from the front of the list and returns it. Returns None if the list is empty. |
| Is\_full | public | self | bool | Returns if the list is full. |
| Is\_empty | public | self | bool | Returns if the list is empty. |
| \_\_str\_\_ | public | self | str | Prints each element of the queue separated by whitespace. |

### Networking.py

#### BaseServer(ThreadingHTTPServer)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from ThreadingHTTPServer |
| Self.routing\_methods | public | Dict[str, type] | Refers URL suffixes to what type of response they should refer to (e.g. “/ping”: PingRequest) |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, server\_address, request\_handler\_class | None | Creates a Threading HTTP Server on a given IP address, port tuple using a given request\_handler\_class. |
| start | public | self | None | Starts the server. |
| Stop | public | self | None | Stops the server. |
| Thread\_start | public | self | Threading.Thread | Starts the server on a specific thread that is returned – this likely performs the same function as start() now that ThreadingHTTPServer is used, instead of HTTPServer. |
| Thread\_stop | public | Self, thread | None | Stops the server and the thread it was running on. |

#### HTTPSubnetRequestHandler(BaseHTTPRequestHandler)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from BaseHTTPRequestHandler |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| Common\_request\_handler | protected | Self, method\_name, common\_request, node | None | Handles a request given to it, by calling the corresponding “server\_” method in Node, and then sending it as a pickled response 200 object via HTTP. If anything fails, and ErrorResponse dictionary is generated instead, pickled and sent as a response 400. |
| Do\_POST | public | self | None | Handles an HTTP “POST” request, by depickling, then interpreting the body into a CommonRequest object. The subnet value, which is not included by CommonRequest, is compared to self.server.subnets, to get node, which is passed to common\_request\_handler. |

#### TCPSubnetServer(BaseServer)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from BaseHTTPRequestHandler |
| subnets | public | Dict | Contains integer subnet values which route to different nodes. This allows for easier testing on the same IP-port. |
| Routing\_methods | public | Dict[str, type] | Refers URL suffixes to what type of response they should refer to (e.g. “/ping”: PingSubnetRequest) |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, server\_address | None | Creates HTTP Server |

#### HTTPRequestHandler(BaseHTTPRequestHandler)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from BaseHTTPRequestHandler |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| Common\_request\_handler | protected | Self, method\_name, common\_request, node | None | Handles a request given to it, by calling the corresponding “server\_” method in Node, and then sending it as a pickled response 200 object via HTTP. If anything fails, and ErrorResponse dictionary is generated instead, pickled and sent as a response 400. |
| Do\_POST | public | self | None | Handles an HTTP “POST” request, by depickling, then interpreting the body into a CommonRequest object. The data is then sent to the common request handler. |

#### TCPServer(BaseServer)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| node | public | Node, where Node.protocol is a “TCPProtocol” | The node that runs this TCPServer. The URL and port associated with the nodes protocol is used to boot the server. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, node | None | Creates a server using TCP, based on a Threading HTTP Server from http.server, the given node provides the IP and port tuple to start the server. |

### Node.py

#### Node

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| Our\_contact | public | Contact | This nodes contact. |
| storage | public | IStorage | Storage object where key-value pairs will be stored. |
| Cache\_storage | public | IStorage | Short term storage. |
| dht | public | DHT | The DHT object which owns this node, this is set after the node is created by the master DHT object. |
| Bucket\_list | public | BucketList | The list of buckets managed by this Node, this contains all contacts this node knows. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, contact, storage, cache\_storage | None | Initialises the node object. This object is used to represent a device on the network – contains IP, port, ID, storage, cache\_storage, and the master DHT object controlling it. |
| ping | public | Self, sender | Contact | Someone is pinging us. Register the contact and respond with our contact. |
| store | public | Self, key, sender, val, is\_cached, expiration\_time\_sec | None | Stores a key-value pair in main/cache storage.  This adds the sender to our bucket list – so it will error if the sender is ourselves. Then it will send key values if the contact is new (and not is\_cached), then it will save the key-value pair in the corresponding storage object. |
| Find\_node | public | Self, key, sender | Tuple[list[contact], str] | Finds K close contacts to a given ID whilst excluding the sender. It also adds the contact to our bucket list, and sends key values if it is a new contact.  It returns the contacts. |
| Find\_value | public | Self, key, sender | Tuple[list[contact] | None, str | None] | Sends key values if new contact, then attempts to find the value of a key-value pair in our storage (then cache storage), given the key. If it cannot do that, it will return K contacts that are closer to the key than it is. |
| Send-key-values-if-new-contact | public | Self, sender | None | If the contact is new we will check its ID against all of our key-value pairs, if it is one of the K closest on a given key-value, we will send a STORE RPC to it to make it store the key-value. |
| Is\_new\_contact | protected | Self, sender | bool | Returns NOT(if the contact exists in our bucket list or in our DHT’s pending contact list.) |
| Simply\_store | public | Self, key, val | None | For testing purposes:  Sets key-value pair in main storage without any checking. |
| Server\_ping | public | Self, request | dict | Formats incoming request and performs ping(). Replies with the identical random\_id. |
| Server\_store | public | Self, request | dict | Formats incoming request and performs store() Replies with the identical random\_id. |
| Server\_find\_node | public | Self, request | dict | Formats incoming request and performs find\_node(). Replies with the identical random\_id, and the contacts from find\_node(). |
| Server\_find\_value | public | Self, request | dict | Formats incoming request and performs find\_value(). Replies with the identical random\_id, and the contacts + value from find\_value(). |

### Protocols.py

#### VirtualProtocol(IProtocol)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| response | public | bool | If the protocol will respond to requests – this is for testing purposes and is enabled by default. |
| node | public | Node | The master node of this protocol. |
| type | public | str | “VirtualProtocol” used for specifying what type of protocol is being used when networking. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, node, responds | None | Sets master node, and provides the option to disable responding – however this does not have to be provided. |
| ping | public | Self, sender | RPCError | Pings sender if the protocol responds. Otherwise it will create a timeout RPCError. |
| Find\_node | Public | Self, sender, key | tuple[list[Contact], RPCError] | Finds K close contacts to a given ID, while excluding the sender. It also adds the sender if it hasn't seen it before. |
| Find\_value | public | Self, sender, key, | tuple[list[Contact] | None, str | None, RPCError] | Sends key values if new contact, then attempts to find the value of a key-value pair in our storage (then cache storage), given the key. If it cannot do that, it will return K contacts that are closer to the key than it is. |
| Store | public | Self, sender, key, val, is\_cached, exp\_time\_sec | RPCError | Stores the key-value on the remote peer, and returns no error |

#### TCPSubnetProtocol(IProtocol)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| url | public | Str | IP address used by the protocol |
| port | public | Int | Port used by the protocol |
| responds | public | Bool | If the protocol responds, this is set to True by default. False is only used when testing to make sure timeout errors and unresponding contacts are handled correctly. |
| subnet | public | Int | Which subnet is being used – this corresponds to a certain node serverside, this is used for testing when it is convenient to test a network on 1 IP, 1 port. |
| type | public | str | “TCPSubnetProtocol” used for specifying what type of protocol is being used when networking. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, url, port, subnet | None | Initialises the protocol on a given URL, port and subnet. Responds must be manually set after creating the protocol, so that no errors accidentally occur. |
| Find\_node | public | Self, sender, key | tuple[list[Contact] | None, RPCError] | Encodes all of the data that is needed into a FindNodeSubnetRequest,  Which is then pickled and posted using the ‘requests’ library to self.url and self.port using a “/find\_node” endpoint. This makes sense because our node doesn’t call this – other nodes will call this method to contact us.  A timeout error trips a flag timeout\_error. After formatting the response (if there was one), an RPCError is returned using get\_rpc\_error(), which takes into account the response, timeout\_error, and the random\_id sent. |
| Find\_value | public | Self, sender, key |  | Encodes all of the data that is needed into a FindValueSubnetRequest,  Which is then pickled and posted using the ‘requests’ library to self.url and self.port using a “/find\_value” endpoint. This makes sense because our node doesn’t call this – other nodes will call this method to contact us.  A timeout error trips a flag timeout\_error. After formatting the response (if there was one), an RPCError is returned using get\_rpc\_error(), which takes into account the response, timeout\_error, and the random\_id sent.. |
| ping | public | Self, sender: Contact | RPCError | Encodes all of the data that is needed into a PingSubnetRequest,  Which is then pickled and posted using the ‘requests’ library to self.url and self.port using a “/ping” endpoint. This makes sense because our node doesn’t call this – other nodes will call this method to contact us.  A timeout error trips a flag timeout\_error. After formatting the response (if there was one), an RPCError is returned using get\_rpc\_error(), which takes into account the response, timeout\_error, and the random\_id sent. |
| store | public | Self, sender, key, val, is\_cached, expiration-time-sec | RPCError | Encodes all of the data that is needed into a StoreSubnetRequest,  Which is then pickled and posted using the ‘requests’ library to self.url and self.port using a “/store” endpoint. This makes sense because our node doesn’t call this – other nodes will call this method to contact us.  A timeout error trips a flag timeout\_error. After formatting the response (if there was one), an RPCError is returned using get\_rpc\_error(), which takes into account the response, timeout\_error, and the random\_id sent. |

#### TCPProtocol(IProtocol)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| url | public | Str | IP address used by the protocol. |
| port | public | Int | Port used by the protocol. |
| responds | public | Bool | If the protocol responds, this is set to True by default. False is only used when testing to make sure timeout errors and unresponding contacts are handled correctly. |
| type | public | str | “TCPProtocol” used for specifying what type of protocol is being used when networking. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, url, port, subnet | None | Initialises the protocol on a given URL, port and subnet. Responds must be manually set after creating the protocol, so that no errors accidentally occur. |
| Find\_node | public | Self, sender, key | tuple[list[Contact] | None, RPCError] | Encodes all of the data that is needed into a FindNodeSubnetRequest,  Which is then pickled and posted using the ‘requests’ library to self.url and self.port using a “/find\_node” endpoint. This makes sense because our node doesn’t call this – other nodes will call this method to contact us.  A timeout error trips a flag timeout\_error. After formatting the response (if there was one), an RPCError is returned using get\_rpc\_error(), which takes into account the response, timeout\_error, and the random\_id sent. |
| Find\_value | public | Self, sender, key |  | Encodes all of the data that is needed into a FindValueSubnetRequest,  Which is then pickled and posted using the ‘requests’ library to self.url and self.port using a “/find\_value” endpoint. This makes sense because our node doesn’t call this – other nodes will call this method to contact us.  A timeout error trips a flag timeout\_error. After formatting the response (if there was one), an RPCError is returned using get\_rpc\_error(), which takes into account the response, timeout\_error, and the random\_id sent.. |
| ping | public | Self, sender: Contact | RPCError | Encodes all of the data that is needed into a PingSubnetRequest,  Which is then pickled and posted using the ‘requests’ library to self.url and self.port using a “/ping” endpoint. This makes sense because our node doesn’t call this – other nodes will call this method to contact us.  A timeout error trips a flag timeout\_error. After formatting the response (if there was one), an RPCError is returned using get\_rpc\_error(), which takes into account the response, timeout\_error, and the random\_id sent. |
| store | public | Self, sender, key, val, is\_cached, expiration-time-sec | RPCError | Encodes all of the data that is needed into a StoreSubnetRequest,  Which is then pickled and posted using the ‘requests’ library to self.url and self.port using a “/store” endpoint. This makes sense because our node doesn’t call this – other nodes will call this method to contact us.  A timeout error trips a flag timeout\_error. After formatting the response (if there was one), an RPCError is returned using get\_rpc\_error(), which takes into account the response, timeout\_error, and the random\_id sent. |

### Routers.py

#### BaseRouter

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| Closer\_contacts | Public | List[Contact] | An initially empty list which is updated throughout the course of lookups, containing a list of contacts which are closer to a given node than our node by the XOR metric. |
| Further\_contacts | Public | List[Contact] | An initially empty list which is updated throughout the course of lookups, containing a list of contacts which are farther to a given node than our node by the XOR metric. |
| Node | Public | Node | The node associated with the Router – this contains: ID, Protocol, bucket list etc. |
| dht | public | DHT | The master DHT controlling this Router, this is initially “None” but it is set upon DHT initialisation. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, node | None | Initialises the base router, setting closer and further contacts to empty list, and setting self.node. |
| Find-closest-nonempty-bucket | public | Self, key | KBucket | Finds the closest non empty Kbucket in our nodes bucket list to a given key. |
| Rpc\_find\_nodes | public | Self, key, contact | Tuple[list[Contact], None, None] | Performs find nodes () on “contact”, where we are the sender, searching for K nodes near “key”. |
| rpc\_find\_value | public | Self, key, contact | Tuple[list[Contact], Contact, str] | Performs find value() on “contact”, where we are the sender searching for a value corresponding to “key”, or K contacts close to “key”, if we find the value, we will also receive the contact that contains it. |
| query | protected | Self, key, nodes\_to\_query, rpc\_call, closer\_contacts, further\_contacts | FindResult | Gets nodes that are closer to “key” than a node we are querying – we query all of nodes-to-query. This ends as soon as we find a value. A FindResult object is then returned containing closer\_contacts, found, found\_by, and the value we found. |
| lookup | public | Self, key, rpc\_call, give\_me\_all | FindResult | Abstract method: Performs main Kademlia lookup |
| Get\_closer\_nodes | public | Self, key, node\_to\_query, rpc\_call, further\_contacts, closer\_contacts | tuple[bool, str, Contact, list[Contact], list[Contact]] | Gets nodes that are closer to “key” than “node\_to\_query”.  Gets nodes by performing rpc-call on the node to query, looking for “key”, if we don’t know it, it is added to a list “peers\_nodes”.  Checking each contact in peers\_nodes, we check if it is closer to the key than the node\_to\_query, if it is, we add it to closer\_contacts. We do a similar check to check if it is further, and then we add it to further\_contacts. |

#### Router(BaseRouter)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from BaseRouter. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, node | None | Initialises router object, this does not use threading and insteads relies on a serial approach, which is rather slow. It inherits most from BaseRouter. |
| lookup | public | Self, key, rpc\_call, give\_me\_all | FindResult | Give me all is by default set to False. This performs the main Kademlia lookup algorithm serially.  This method initiates a Kademlia lookup operation, searching for nodes closest to the given key. It starts by getting nodes from the closest non-empty k-bucket and continues querying nodes serially until it has gathered responses from the closest k nodes or until we run out of nodes to contact. |

#### ParallelRouter(BaseRouter)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| … | … | … | Inherited from BaseRouter, a parameter “node” is passed to BaseRouter. |
| Contact\_queue | Private | InfiniteLinearQueue | Queue of “contact queue items” dictionarys that haven’t been handled. |
| Semaphore | private |  | Ensures thread-safe access to shared resources between all MAX-THREADS in the ParallelRouter, it does this by sending “release” and “acquire” commands. “release” increments the counter by 1 – saying that there is work for threads, waking up a waiting thread if the counter was 0 (there was 0 work previously). |
| now | private | datetime | Current time. |
| Stop\_work | private | bool | If the ParallelRouter should stop all work it is doing, set to False by default. |
| threads | private | List[threading.Thread] | List of all threads being used by the ParallelRouter. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| Initialise-thread-pool | private | self | None | Creates and starts MAX-THREADS # of threads, which all run self.rpc-caller. |
| Queue\_work | public | Self, key, contact, rpc-call, closer-contacts, further-contacts, find-result | None | Adds new Contact Queue Item to self.\_\_contact-queue, all of the parameters listed are added. The semaphore is released at the end of this function to signal that there is work available in the queue. |
| Rpc\_caller | private |  |  | This is ran on each thread in parallel inside the Router; it is an infinite loop that exists for as long as the Router is running. It will check for work from the semaphore – if there is no work it will wait until there is. Once there is work it will dequeue an item from the queue and get K nodes closer to “key” than “contact”, updating the FindResult – this works even though it has been dequeued because python refers to lists as references (for example, if you pass a list into a function, the function can edit the list, and that will persist outside the function), so we can refer to item[“findResult”] because the reference will persist in the “lookup” method. |
| Set-query-time | public | self | None | Sets self.now() to current time. |
| Query-time-expired | protected | self | bool | Returns if the time since query was triggered is longer than Constants REQUEST-TIMEOUT. |
| Dequeue-remaining-work | private | self | None | Dequeues everything from the contact queue. |
| Stop-remaining-work | protected | self | None | Dequeues everything from the contact queue, then sets “stop work” to True, so no more work will be done. |
| Parallel\_found | public | Self, find\_result, find\_ret | Tuple[bool, FindResult] | Overlays found-ret with find-result if find-result has a value. It then returns if the overlay has been performed, and the new found\_ret. |
| lookup | public | Self, key, rpc\_call, give\_me\_all | FindResult | Give\_me\_all is false by default.  Performs a similar algorithm to what has been performed in "Router", but this time it is performed in parallel. First it makes sure that self.node actually exists to prevent any strange errors from happening down the line. It then gets a list of K 'all nodes' it intends to query, and then bits of chunks of ALPHA each time and queries them individually. It then groups these ALPHA nodes\_to\_query into closer\_contacts  and further\_contacts, depending on if they are closer than or further than our ID to the parameter "key" by the XOR metric. All of the closer and further contacts are then placed into a ContactQueueItem (and appended to our contact queue) with each member of the nodes to query - This will be handled by the Constants.MAX\_THREADS running rpc\_caller. The time since last query is then updated. This process then iterates, biting of chunks of ALPHA contacts until there are no closer or further uncontacted nodes, or until one of the threads handling the contact queue finds the value we are looking for which matches the key-value pair with "key", if that is the case, the FindResult object containing the value will be returned. |

### Storage.py

#### VirtualStorage(IStorage)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| store | protected | Dict[int, StoreValue] | A simple storage device in memory, which is just a dictionary storing key-value pairs. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | self | None | Creates an empty storage dictionary. |
| contains | public | Self, key | bool | Returns a Boolean stating whether a key-value pair exists, given key. |
| get | public | Self, key | str | Returns stored value associated with given key. |
| Get\_timestamp | public | Self, key | datetime | Returns when the key was last republished as a datetime object. |
| set | public | Self, key, value, expiration-time-sec | None | Stores a key value pair, along with the expiration time and timestamp. |
| Get-expiration-time-sec | public | Self, key | int | Returns how long it takes for the given key-value pair to expire, given key. |
| remove | public | Self, key | None | Removes a given key-value pair, given key. |
| Get\_keys | public | Self | List[int] | Returns all keys of key-value pairs that are stored. |
| touch | public | Self, key | None | “touches” a given key-value pair, this is done by updating the timestamp to the current time. |
| Try\_get\_value | public | Self, key | Tuple[bool, str] | Tries to get a given value from a key-value pair, given the key. Returns True | False, and the value if it was found. |

#### SecondaryJSONStorage(IStorage)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| filename | public | str | The filename the storage is saved to. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, filename | None | Sets the filename to store the data at. |
| set | public | Self, key, value, expiration-time-sec | None | Sets a key-value pair in the JSON along with the expiration time in seconds, and the timestamp as the current time. The python JSON library cannot store datetime objects, so it is converted in and out of “ISOFormat” which is a string representation of it. |
| contains | public | Self, key | bool | Returns if the storage file contains a key-value pair, given the key. |
| Get\_timestamp | public | Self, key | datetime | Gets the timestamp of a key-value pair, given the key. |
| get | public | Self, key | str | Returns the value of a key-value pair from the storage file, given the key. |
| Get-expiration-time-sec | public | Self, key | int | Gets the time to expire for a key-value pair, given the key. |
| remove | public | Self, key | None | Removes a key-value pair, given the key. |
| Get\_keys | public | Self | List[int] | Returns all keys stored by the storage file as a list of integers. |
| touch | public | Self, key | None | “touches” a key-value pair by setting the timestamp to the current time. |
| Try\_get\_value | public | Self, key | Tuple[bool, str] | Tries to get a given value from a key-value pair in the storage file, given the key. Returns True | False, and the value if it was found. |
| Set\_file | public | Self, key, filename, expiration-time-sec | None | Adds a file to storage file, it does this by loading ALL of the file to be added to memory, and then pasting it into the storage file ALSO loaded into memory. This is extremely memory inefficient. |

### GUI.py

#### Fonts

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| Title\_font | public | Tuple | The tuple of font I want to use for any titles in my GUI |
| Text\_font | public | tuple | The tuple of font I want to use for any text in my GUI. |

#### ContactViewer(customtkinter.CTk)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| Id | Public | Int | The ID we are viewing |
| url | Public | Str | The IP we are viewing |
| Port | Public | Int | The port we are viewing |
| Protocol\_type | Public | Type | The type of protocol the contact we are viewing uses. |
| Contact-viewer-title | Public | CTk.CTkLabel | Title of window |
| Id\_label | Public | CTk.CTkLabel | The ID we are viewing in text on the screen |
| Protocol\_type\_label | Public | CTk.CTkLabel | The type of protocol we are viewing as text on the screen |
| url\_label | Public | CTk.CTkLabel | The IP we are viewing as text on the screen |
| Port\_label | Public | CTk.CTkLabel | The port we are viewing as text on the screen. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, id, protocol\_type, url, port, appearance\_mode | None | Creates the contact viewer, appearance\_mode is “dark” by default but “light” is valid too. |
| Show\_error | public | Self, error\_message | None | Shows the error window, displaying the given error message. |
| Export\_contact | public | Self, filename | None | Filename is “our\_contact.json” by default. |
| Show\_status | public | Self, message | None | Shows a status window, displaying the given message. |

#### StatusWindow (customtkinter.CTk)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| copy\_data | public | str | Data to be copied to the clients clipboard. |
| message | public | Ctk.CTkLabel | Message displayed to the client. |
| Copy\_button | Public | Ctk.CTkButton | Button to copy copy\_data to the clients clipboard. This is only displayed if there is copy data. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, message, copy\_data | None | Creates the status window, with option for copying data to clipboard if there is copy data. |
| copy | public | self | None | Copys self.copy\_data to clipboard. |

#### Settings (customtkinter.CTk)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| Appearance\_mode | public | str | The appearance mode of the window (light/dark) |
| dht | public | DHT | The DHT object used by us. |
| Settings\_title | public | Ctk.CTkLabel | The title of the window. |
| Dht\_export\_label | public | Ctk.CTkLabel | Label asking the user where they would like DHT to be exported to. |
| Export\_dht\_button | public | Ctk.CTkButton | Button to Export/save our DHT to a file. |
| View\_contact\_button | public | Ctk.CTkButton | Button to view our contact – this opens a ContactViewer window. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, hash\_table, appearance\_mode | None | Creates the settings window using an existing DHT object, appearance-mode is ”dark” by default. |
| Show\_error | public | Self, error\_message | None | Creates a new error window, displaying the given error message. |
| Show\_status | public | Self, message | None | Creates a new status window, displaying the given message. |
| Export\_dht | public | self | None | Exports the DHT to the file provided by dht\_export\_file, handling the exception and raising a new error window displaying the failure reason. |
| View\_contact | public | self | None | Creates new ContactViewer window, displaying the contact used by our DHT object. |

#### ErrorWindow (customtkinter.CTk)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| title | public | Ctk.CTkLabel | Title of error window |
| Error\_message | public | Ctk.CTkLabel | Error message |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, error\_message | None | Creates error window with given error message. |

#### MainGUI (customtkinter.CTk)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| Appearance\_mode | public | str | Theme of the window. This is “dark” by default, can also be “light”. |
| dht | public | DHT | Our DHT |
| server | public | TCPServer | The server we are using to handle requests. |
| Server\_thread | public | Threading.Thread | The thread the server is running on. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, appearance-mode | None | Creates the mainGUI object by creating the “Join DHT” frame inside itself. |
| Initialise-kademlia | public | self | None | Creates DHT, server and server thread. If GET-GLOBAL-IP is true, then it will get our global IP by decoding a response from ‘https://api.ipify.org’, which according to a StackOverflow article is the most efficient way to get your global IP in python. If GET-GLOBAL-IP is false, IP is set to “127.0.0.1”. This is useful for DHTs on just the local networks, so no port forwarding needs to be set up.  Then a valid port is attempted to be found by getting a random integer between 5000 and 35000, until the port is free. This allows for multiple instances on one device, because the port is not hard coded. A TCPProtocol is created with this IP and Protocol. Then a contact is created with a random ID, and the protocol we just created. A JSON storage object is setup for main storage, and VirtualStorage for cache storage. These are placed into a subfolder with title “id.value”.  Now we have initialised Kademlia, the main network frame is launched. |
| Open\_settings | public | self | None | Opens the settings window in a new window. |
| Thread-open-settings | public | self | None | Opens the settings window in a new window on a new thread. |
| Add-settings-icon | public | self | None | Adds the settings button/icon to the bottom middle of the window. |
| Clear\_screen | public | self | None | Destroys all children of the current window |
| Clear\_screen-and-keep-settings | public | self | None | Destroys all children of the current window, keeping the settings icon. |
| Make-join-dht-frame | public | self | None | Makes the Join DHT frame. |
| Make-load-dht-frame | public | self | None | Makes the Load DHT frame. |
| Make-bootstrap-frame | public | self | None | Makes the bootstrap frame. |
| Make-bootstrap-from-json-frame | public | self | None | Makes the “Bootstrap from file” frame. |
| Make-network-frame | public | self | None | Makes the main network frame – this is the one that says “download” and “upload”. |
| Show\_error | Public | Self, error\_message | None | Displays error window with error message. |
| Show\_status | public | Self, message, copy\_data | None | Displays new window displaying status message, and an option to copy some data to clipboard. |
| Make-download-frame | public | self | None | Makes the “download file” frame. |
| Make-upload-frame | public | self | None | Makes the “upload file” frame. |

#### UploadFrame(customtkinter.CTkFrame)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| parent | Public | MainGUI | The main customtkinter window which owns this frame instance. |
| title | public | Ctk.CTkLabel | Title of frame: “Upload File” |
| Enter\_file\_label | public | CTkLabel | Label prompting the user to enter the file they would like to upload to the network. |
| Enter\_file\_textbox | public | Ctk.CTkTextbox | The textbox where the user enters the file/path to the file of the file they would like to upload. |
| Back\_button | public | Ctk.CTkButton | Goes back to main network window. |
| Upload\_button | public | Ctk.CTkButton | Triggers handle\_upload() |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, parent, fg\_colour, \*\*kwargs | None | Creates a CTkFrame in “parent” with keyword arguments “\*\*kwargs”, and then foreground colour is set to “transparent” by default. The frame contains all information to upload a file to the network. |
| Handle\_upload | public | self | None | Reads file\_to\_upload from enter\_file\_textbox. If it is a file, We save the file as a key-value pair, where the key is a random ID, and the value is a pickled dictionary containing the base filename and the file contents themselves. |

#### DownloadFrame(customtkinter.CTkFrame)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| parent | Public | MainGUI | The main customtkinter window which owns this frame instance. |
| title | public | Ctk.CTkLabel | Title of frame: “Download File” |
| Enter\_id\_label | public | CTkLabel | Label prompting the user to enter the key of the key-value they would like to download from the network. |
| Enter\_id\_textbox | public | Ctk.CTkTextbox | The textbox where the user enters the key of the key-value the would like to download from the network. |
| Back\_button | public | Ctk.CTkButton | Goes back to main network window. |
| download\_button | public | Ctk.CTkButton | Triggers handle\_download() |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, parent, fg\_colour, \*\*kwargs | None | Creates a CTkFrame in “parent” with keyword arguments “\*\*kwargs”, and then foreground colour is set to “transparent” by default. The frame contains all information to download a file from the network. |
| Handle\_download | public | self | None | Tries to download a file from the network given an ID. It checks the ID is valid, then converts it into an ID object. Then the parents DHT calls find-value on the validated ID given by the user.  If it wasn’t found, an error window is created saying that the file was not found.  If it was found, the value received from the network is a pickled dictionary containing the base filename and file data. The file is saved to the current working directory as the filename from the network. Then a status window is created saying it was a success. |

#### MainNetworkFrame(customtkinter.CTkFrame)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| parent | Public | MainGUI | The main customtkinter window which owns this frame instance. |
| title | public | Ctk.CTkLabel | Title of frame: “Download File” |
| Download\_button | Public | Ctk.CTkButton | Creates download file frame. |
| Upload\_button | public | Ctk.CTkButton | Creates upload file frame. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, parent, fg\_colour, \*\*kwargs | None | Creates a CTkFrame in “parent” with keyword arguments “\*\*kwargs”, and then foreground colour is set to “transparent” by default. The frame allows the user the option of downloading or uploading a file. |

#### LoadDHTFromFileFrame(customtkinter.CTkFrame)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| parent | Public | MainGUI | The main customtkinter window which owns this frame instance. |
| title | public | Ctk.CTkLabel | Title of frame: “Load DHT from File” |
| Enter\_filename\_text | public | CTkLabel | Label prompting the user to enter the filename of the dht object they would like to load. |
| filename\_textbox | public | Ctk.CTkTextbox | The textbox where the user enters the filename of the DHT they would like to load. |
| Back\_button | public | Ctk.CTkButton | Goes back to main network window. |
| Load\_button | public | Ctk.CTkButton | Triggers load\_dht() |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, parent, fg\_colour, \*\*kwargs | None | Creates a CTkFrame in “parent” with keyword arguments “\*\*kwargs”, and then foreground colour is set to “transparent” by default. The frame allows the user to load a DHT from a file, or return back to the menu. |
| Load\_dht | Public | self | None | Reads filename from textbox, validates it. If valid, a new DHT object will be loaded. Any existing DHTs and servers are replaced and shutdown. Any error when making the new server creates a new error window displaying the error.  If there was not an error, the main network frame is called. |

#### JoinNetworkMenuFrame(customtkinter.CTkFrame)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| parent | Public | MainGUI | The main customtkinter window which owns this frame instance. |
| Load\_button | Public | Ctk.CTkButton | Button which triggers parent DHT to create the “load DHT” frame |
| Bootstrap\_button | Public | Ctk.CTkButton | Button which triggers parent DHT to create the bootstrap frame. |
| Create-new-network-button | Public | Ctk.CTkButton | Button which triggers parent DHT to run “initialise\_kademlia” |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, parent, fg\_colour, \*\*kwargs | None | Creates a CTkFrame in “parent” with keyword arguments “\*\*kwargs”, and then foreground colour is set to “transparent” by default. The frame allows the user to have the choice on their manner of joining the network. |

#### BootstrapFromJSON(customtkinter.CTkFrame)

##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| parent | Public | MainGUI | The main customtkinter window which owns this frame instance. |
| title | public | Ctk.CTkLabel | Title of frame: “Bootstrap from Contact JSON” |
| Filename\_entry | public | CTk.CTkEntry | Allows the user to enter the file they wish to bootstrap from. |
| Back\_button | public | Ctk.CTkButton | Goes back to main network window. |
| Load\_button | public | Ctk.CTkButton | Attempts to bootstrap from JSON by calling load-known-peer-json-for-bootstrap. |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, parent, fg\_color, \*\*kwargs | None | Creates a CTkFrame in “parent” with keyword arguments “\*\*kwargs”, and then foreground colour is set to “transparent” by default. The frame allows the user to bootstrap into a Kademlia network from an existing contact JSON file. |
| Load-known-peer-json-for-bootstrap | public | Self | None | Gets the filename from self.filename-entry, validates it, creating an error window if the file is invalid.  Otherwise, the dictionary of contact information is loaded from the JSON file using the “json” library. It then validates the dictionary, raising error windows if it was deformed. Then BootstrapFrame Boostrap() is called using the contact dict. |

#### BootstrapFrame(customtkinter.CTkFrame)

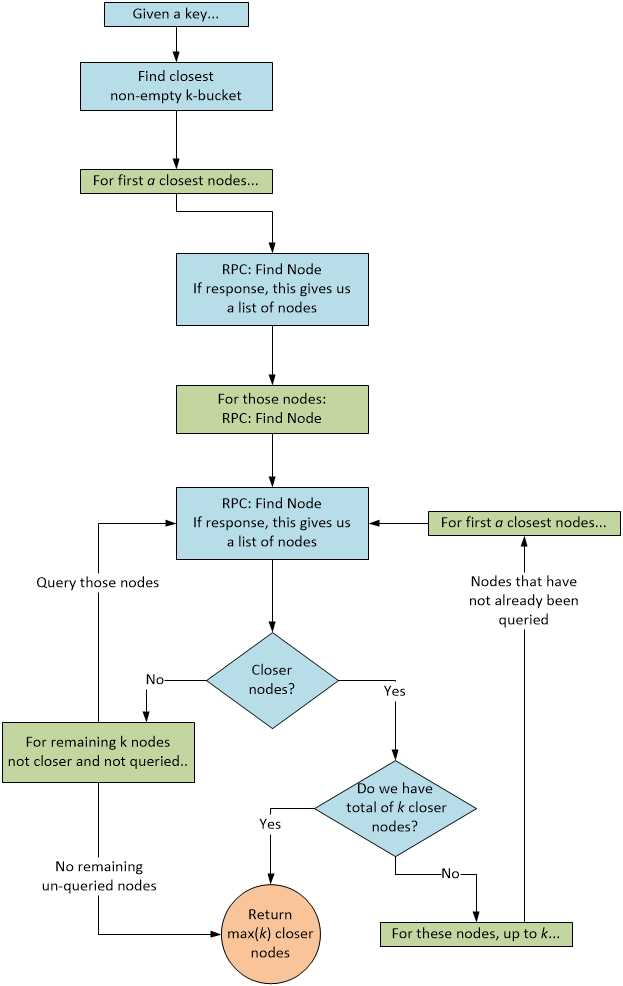
##### Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Visibility | Type | Description |
| parent | Public | MainGUI | The main customtkinter window which owns this frame instance. |
| title | public | Ctk.CTkLabel | Title of frame: “Bootstrap from known peer” |
| ip\_entry | public | CTk.CTkEntry | Allows the user to enter the IP they wish to bootstrap from. |
| port\_entry | public | CTk.CTkEntry | Allows the user to enter the port they wish to bootstrap from. |
| id\_entry | public | CTk.CTkEntry | Allows the user to enter the ID they wish to bootstrap from. |
| Back\_button | public | Ctk.CTkButton | Goes back to main network window. |
| Load\_from\_json\_button | public | Ctk.CTkButton | Allows the user to load from a contact JSON instead (calls “make bootstrap from json frame” in parent) |
| connect\_button | public | Ctk.CTkButton | Allows the user to connect using the data they just provided. (this calls handle\_bootstrap) |

##### Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Visibility | Parameters | Returns | Description |
| \_\_init\_\_ | N/A | Self, parent, fg\_colour, \*\*kwargs | None | Creates a CTkFrame in “parent” with keyword arguments “\*\*kwargs”, and then foreground colour is set to “transparent” by default. The frame allows the user to bootstrap off a known peer by manually inputting its information. |
| Handle\_bootstrap | Public | self | None | Gets IP from ip\_entry, then validates it against a Regex string. The Port is read from port\_entry, which is validated to ensure it is numeric between 0 and 65535. The ID is read from id\_entry and validated to ensure it is numeric between 0 and 2^ID-LENGTH-BITS. |
| Bootstrap | public | Cls, parent, known\_id, known\_ip, known\_port | None | CLASS METHOD: Bootstraps into Kademlia network from known peer. This is a class method because BootstrapFromJSON uses it too.  This creates a new TCPProtocol from the known\_url and known\_port, which is used to create the known peer contact, out of that protocol and the known\_id. Then we bootstrap using the parent windows DHT. |

## Node Lookup



From the flowchart above (<https://www.syncfusion.com/succinctly-free-ebooks/kademlia-protocol-succinctly/node-lookup>), a rough framework for how to look up nodes can be established.

## Creating Kademlia

Kademlia is an extremely complicated protocol; I will be using the [original specification](https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf), and the book “[Kademlia Protocol Succinctly](https://www.syncfusion.com/succinctly-free-ebooks/kademlia-protocol-succinctly/)” by Marc Clifton to help me with my project. This book contains a variety of flowcharts, diagrams and C# code snippets to explain this protocol.

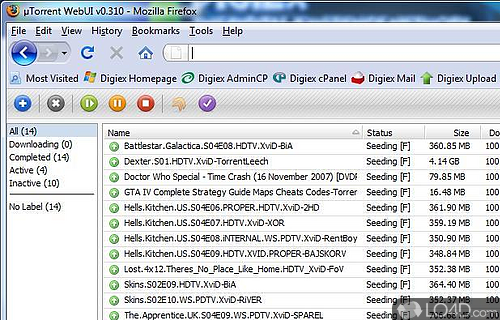
## User Interface Design

A paper with writing on it

Description automatically generated

I also would like this to create pop up windows when something major has been completed or if the user inputs something incorrect.

I would like to have a download/upload window like uTorrent:



This may prove too ambitious though.

I would like the UI to start on the “**Join a network**” page, and then the page will update when each button is pressed. There is also a settings cog which can be opened and should make the “**Settings**” Window as you can see bottom right.

## Storing Files

Since Kademlia uses Key-Value pairs to store references to files – or the files themselves, I would like to stay truthful to the spirit of Kademlia, so I would like to make as few extra methods as possible. Node.get\_value() gets the value from storage using key value pairs.

Do I write the files being shared on the Kademlia system to a specific Kademlia storage file, or do I write references to the file paths into a Kademlia storage in memory?

Example storage object which uses file paths:

|  |  |
| --- | --- |
| ID | File path |
| 1184636153678088884358269235741455443449620727280 | “C:\Windows\system32” |
| 903630858704281796858737593619265100499384915413 | “D:\My Games” |
| 929756209273338579416183776433308233718260102263 | “C:\Windows\Users\gabri” |
| 577515970642880934129029851154338854323671305539 | “D:\Homework” |

This would allow for a file to be dynamically updated.

However, that comes with the downside that it is then impossible to propagate a file across the network, as a node which does not share the file path could store it, and then attempt to access it. This could be fixed by appending IP and port number to the file path, like this:

|  |  |
| --- | --- |
| ID | File path with IP + Port |
| 1184636153678088884358269235741455443449620727280 | {“url”: “81.140.113.37”, “port”: 3245, “file path”: “C:\Windows\system32”} |
| 903630858704281796858737593619265100499384915413 | {“url”: “81.140.113.37”, “port”: 3245, “file path”: “D:\My Games” } |
| 929756209273338579416183776433308233718260102263 | {“url”: “81.140.113.37”, “port”: 3245, “file path”: “C:\Windows\Users\gabri”} |
| 577515970642880934129029851154338854323671305539 | {“url”: “81.140.113.37”, “port”: 3245, “file path”: “D:\Homework”} |

This would defeat the point of having a DHT in the first place, because there would be a singular point of failure and no value propagation across the network. Also this just looks like an extremely bad database, so I don’t want to use that.

The other alternative is that I place the raw file as the value. This would make seeing download progress extremely difficult client side and server side, because get\_value() is propagated by placing the encoded value in an HTTP request, and sending it as the body.

This raises the issue of maximum file size, with my current implementation. This is because my implementaton uses ”VirtualStorage”, which stores the object as a dictionary in memory, which leads to the issue of limited heap space – so the maximum file size is limited by the heap. Because of this, I will have to create a new ”SecondaryStorage” object which stores the value as a JSON file. This does not have to be encrypted, as it’s on the client side. There is an ”IJSON” module for python which stands for ’iterative JSON’, which allows for handling of large JSON files; this is useful due to the potentially massive storage object, as people may want to share files that are gigabytes in size. I think uTorrent and BitTorrent separate out files into smaller chunks to prevent issues like this, but that is outside of the scope of this project. **I may have to implement a maximum file size to prevent issues on low-memory machines.**

# TECHNICAL SOLUTION

## What techniques have I used?

|  |  |
| --- | --- |
| Technique | Where |
| Lists | BucketList.buckets |
| Queues | My\_queues.py |
| Complex mathematical/scientific model | Kademlia.py (whole file) – I believe that Kademlia is an incredibly complex scientific model. |
| Graph-tree traversal | Kademlia is arranged in a tree model, as shown by this:  A brief overview of Kademlia and its use in various decentralized platforms  So when find a given node, the tree of nodes is being traversed. Therefore the entire Kademlia protocol is an extremely complex tree searching algorithm, with a depth of 160. |
| Inheritance | **Errors:**  TooManyContactsError inherits Exception  OutOfRangeError inherits Exception  AllKBucketsAreEmptyError inherits Exception  SenderIsSelfError inherits Exception  RPCError inherits Exception  UnknownRequestError inherits Exception  **Dictionary objects:**  FindResult inherits TypedDict  ContactQueueItem inherits TypedDict  BaseRequest inherits TypedDict  FindNodeRequest inherits BaseRequest and TypedDict  FindValueRequest inherits BaseRequest and TypedDict  …  **Interfaces:**  VirtualProtocol inherits IProtocol  VirtualStorage inherits IStorage  TCPSubnetProtocol inherits IProtocol  **Routers:**  Router inherits BaseRouter  ParallelRouter inherits BaseRouter  **Servers:**  TCPServer inherits HTTPServer  TCPSubnetServer inherits HTTPServer |
| Composition | “DHT” has attribute “node” which is composed of a “Node” object |
| Complex client-server model | Incoming requests are handled in:  Networking.TCPServer  Networking.TCPSubnetServer  With HTTP POST calls sent in  Kademlia.TCPSubnetProtocol  Kademlia.TCPProtocol |
| Dictionaries | Most of the dictionaries used are type defined in Dictionaries.py |
| File handling | SecondaryJSONStorage |
| Regex | GUI.BootstrapFrame.handle\_bootstrap |
| HTTP | TCPServer, TCPProtocol, request handlers |
| Parallelisation | semaphores and threads are used in ParallelRouter, I use HTTPThreadingServ |

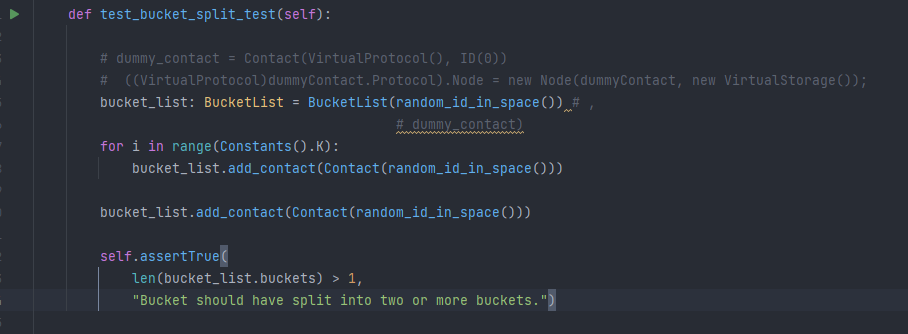
## Reviewing Objectives

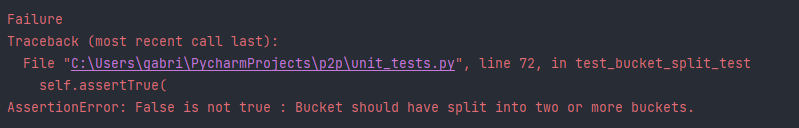
|  |  |  |
| --- | --- | --- |
| Objective name | Met? | Evidence |
| Have a system which can send data between 2 devices using IP and port number. | Yes | Networking.py, protocols.TCPProtocol |
| Check if a device contains the requested data without a central server | Yes | TCPProtocol.find\_value -> Node.find\_value |
| Implementation of KBuckets which split into 2 smaller buckets when full, keeping ID ranges between powers of 2. | Yes | Kbucket.split, Kbucket.add\_contact |
| Unresponsive nodes should be evicted from the network. | Yes |  |
| ID object which can be used interchangeably with integers | Yes |  |
| Be able to send remote procedure call requests to other nodes on the network | Yes |  |
| Method to find nodes on the network | Yes |  |
| Query devices asynchronously | Yes | These tests run a DHT on a ParallelRouter, which queries nodes asynchronously using “threading” and semaphores etc. |
| GUI | Yes |  |
| Data sent must be secure | No |  |
| Should have a dark mode | Yes |  |

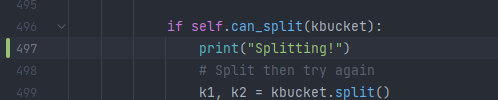
# TESTING

## Debugging

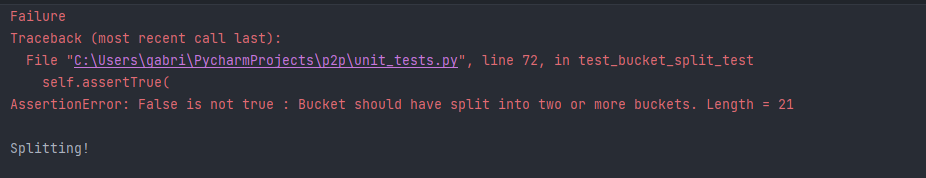
### Problem 1 - Buckets split incorrectly



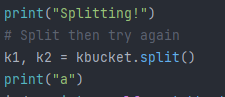




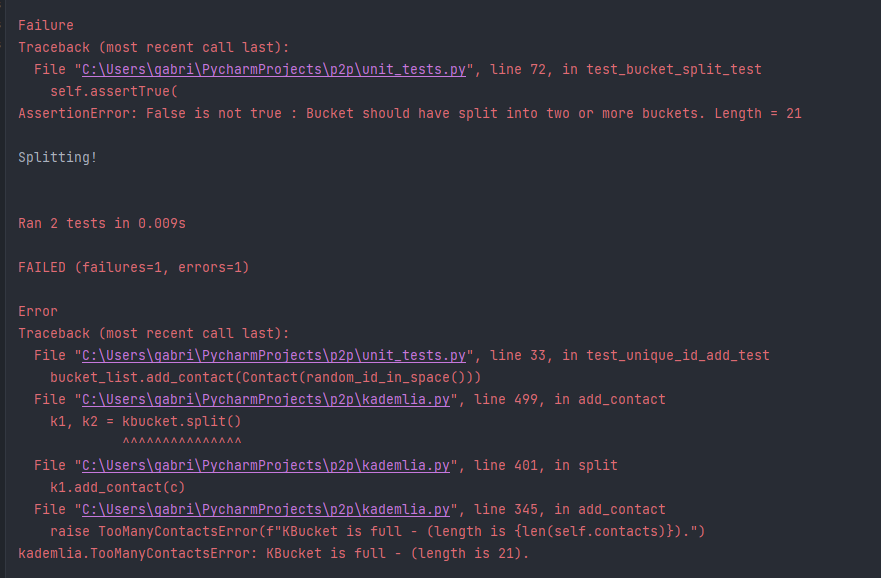
I added a print(”Splitting!”), to make sure that it was able to split, it was:



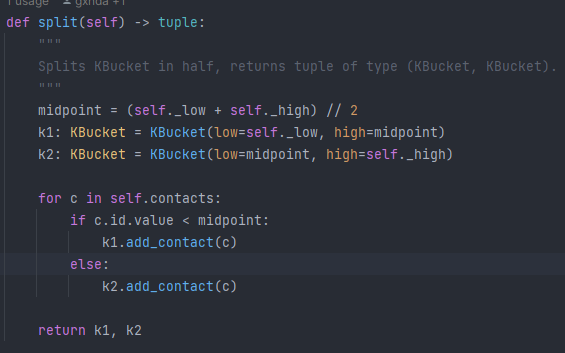
So the issue is that the KBucket knows it should split, but it doesn’t.



With the addition of a print statement, I should be able to check if that point is reached – A breakpoint would be preferable, but I am more comfortable with this more intuitive (albeit simplistic) system.



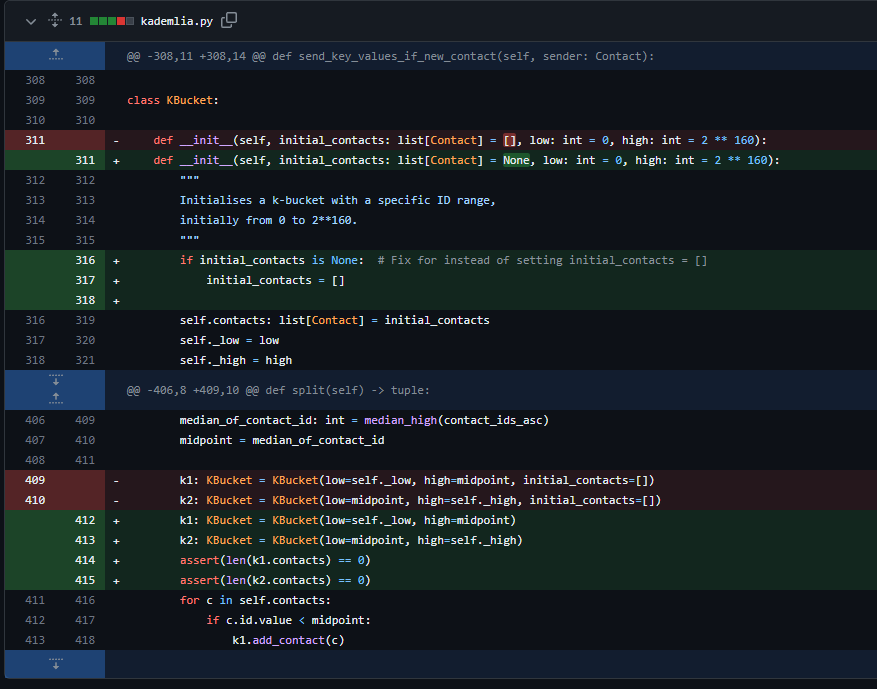
The “a” is not printed, and upon scrolling down, I believe I have found the reason (in hindsight, this is incorrect). In my split() method, the midpoint where it is split is the average between the low and high bounds of the KBucket. This is ineffective against contacts all “bunched up” towards one side of the bucket. Since there is no information on how to create a midpoint in a splitting bucket in the original specification, I am proposing that I use the median of all contact IDs inside the bucket, this will never cause them all to become bunched, as this will only cause a deviation of plus or minus one in size to either side.



*My old split method.*  
  


Upon implementing this. I still got the same error. For some reason, the KBucket() function which creates k1 and k2, was creating itself, both inheriting the original self.contacts (This was seen by testing if self.contacts == k1.contacts, which returned True). This is prevented if I manually set initial\_contacts to be an empty list, but this is not ideal.

This is due to my sloppy KBucket.\_\_init\_\_() function, which initially set initial\_contacts to be an empty list. This caused some strange behaviour, which was very easily fixed:

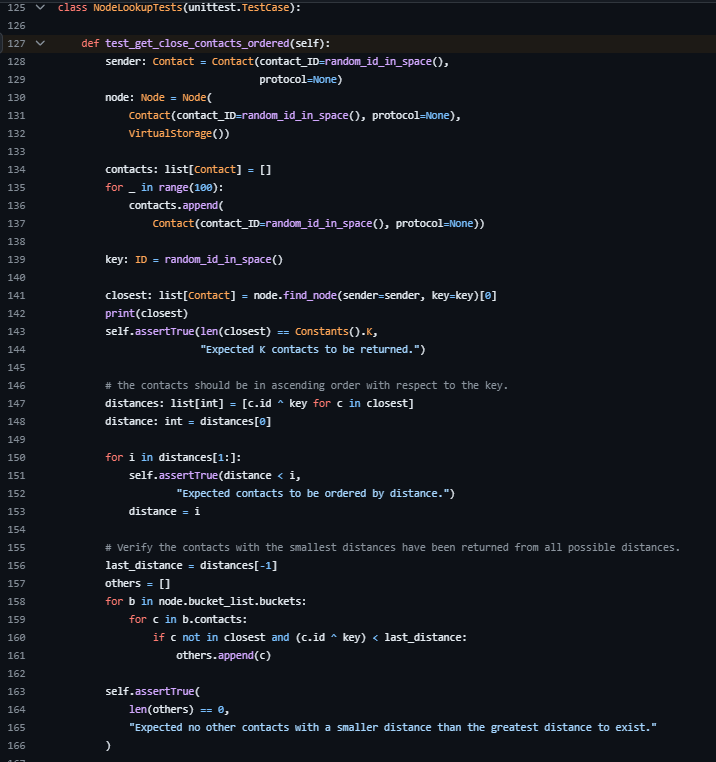


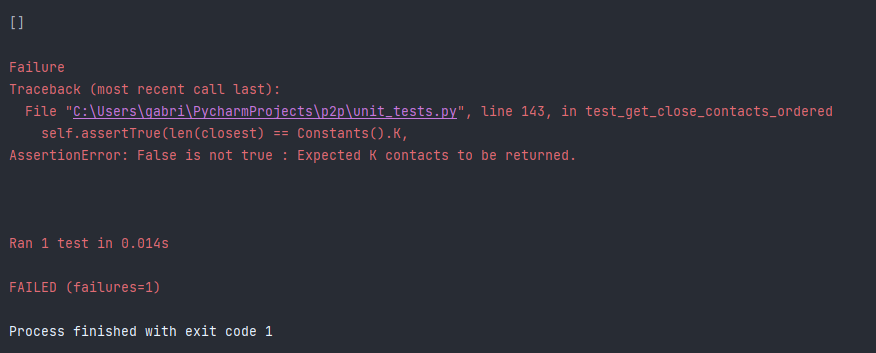
*Above: The fix.*

I removed the initial\_contacts declaration, and I also added some assert() statements to easily identify if the error re-emerges.

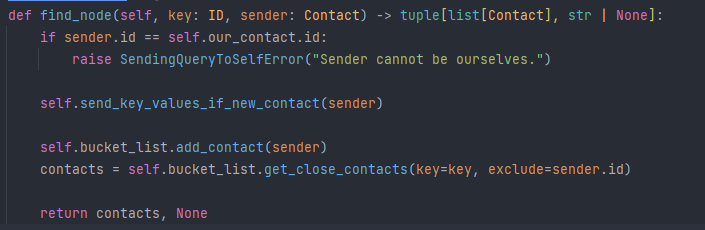
In conclusion, the error was not that the contacts were “bunching together” in the tests, but that the split buckets were inheriting the previous full contact list, causing additional contacts to be unable to be added. I will keep in my updated midpoint for splitting, as it fixes a possible issue which may emerge in some cases.

### Problem 2 – No closest contacts are returned

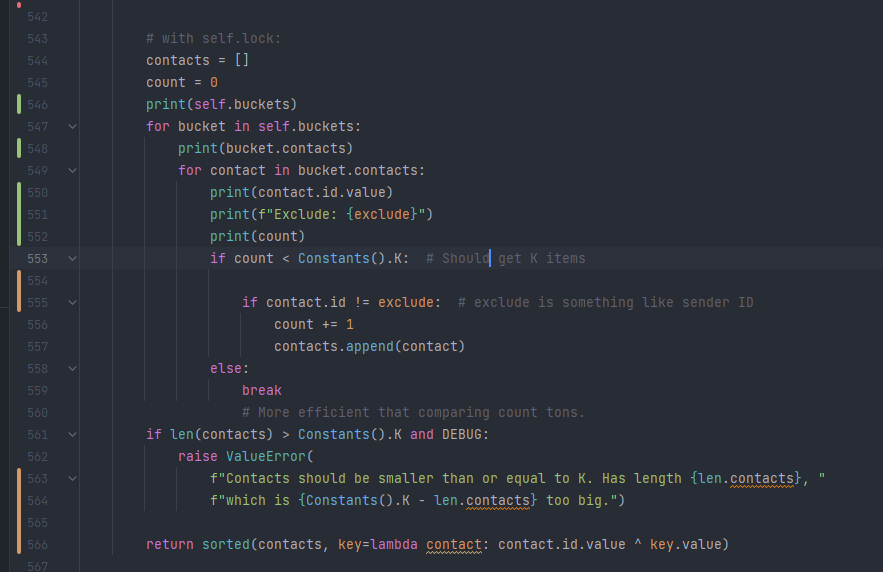


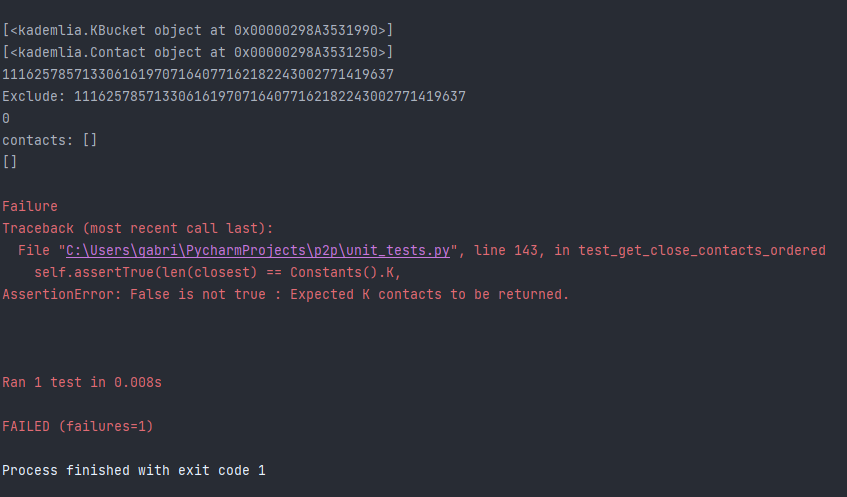


The [ ] is “closest”, this shows me node.find\_node() is not returning anything.

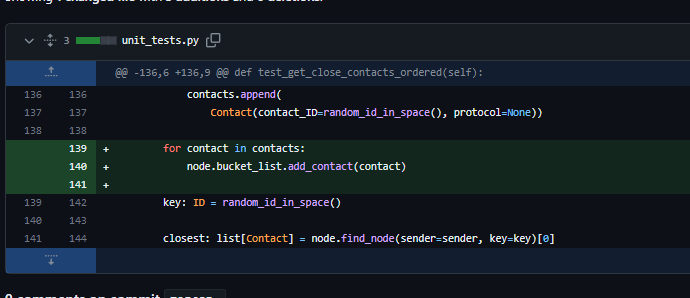


This is node.find\_node(), which should be returning the output of node.bucket\_list.get\_close\_contacts(). I have added several print statements to get\_close\_contacts() to observe whats happening.



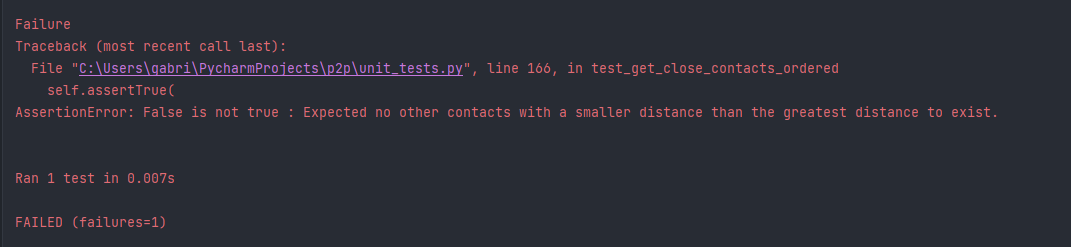


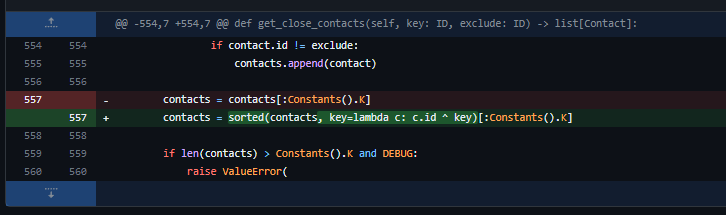
There is only 1 element in the entire bucket list – which is the sender ID (which is being excluded), so of course it does not return anything.



This was a very easy fix; I had just forgotten to add the contacts to the nodes bucket list, so it just required a loop through all contacts, adding them to the bucket list.

### Problem 3 – Unsorted close contacts

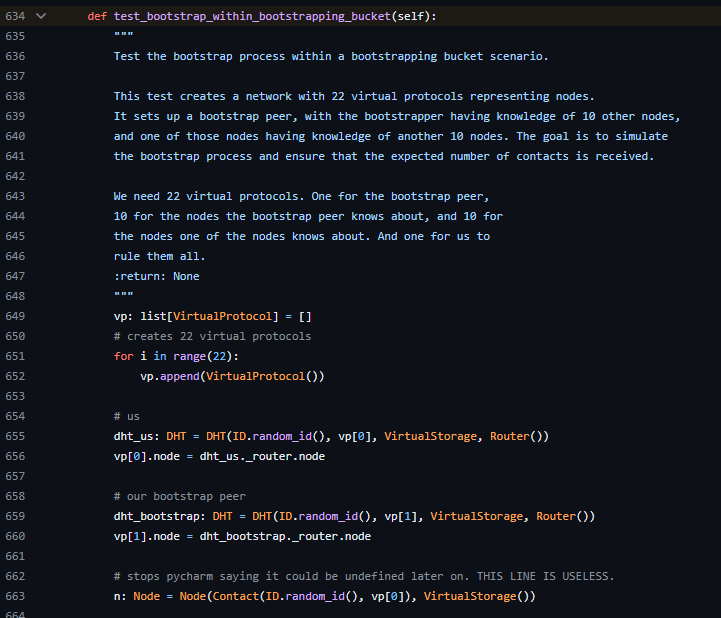




I had forgotten to sort them in BucketList.get\_close\_contacts(), the fix can be seen above.

### Problem 4 – Bootstrapping Issues

A computer screen shot of a program code

Description automatically generated I have 2 tests to test the DHT bootstrapping, one is called “test\_bootstrap\_within\_bootstrapping\_bucket()”, the other is called “test\_bootstrap\_outside\_bootstrapping bucket()”.

A screen shot of a computer

Description automatically generated*Above: test\_bootstrap\_within\_bootstrapping\_bucket().*

*Above: Results of test.*

A screenshot of a computer program

Description automatically generatedThen I have the test “test\_bootstrap\_outside\_bootstrapping\_bucket()”, and I don’t know why it doesn’t work, it is extremely similar to the previous test.  
  
A computer screen shot of a program code

Description automatically generated

*Above: test\_bootstrap\_outside\_bootstrapping\_bucket().*

A screenshot of a computer

Description automatically generated

*Above: result of test.*

After observation, refresh\_bucket() isn’t giving any new contacts.

*A screenshot of a computer program

Description automatically generated*

A screen shot of a computer

Description automatically generated

One of these knows 10 contacts, so this is wrong. This leads me to believe there is an issue with VirtualProtocol.find\_node().

A screen shot of a computer code

Description automatically generated

*Above: VirtualProtocol.find\_node()*

*A computer screen shot of a program code

Description automatically generated*

*Above: Node.find\_node()*

*A screenshot of a computer

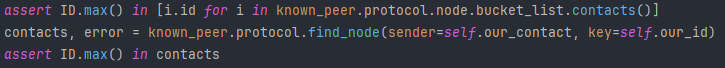
Description automatically generated*

*A computer screen shot of a program

Description automatically generated*

Well, that’s not good.

For some reason the important contact is inside the bootstrappers bucket, when it’s meant to be in one of the contacts inside the bucket’s buckets.





### Ensuring decode/encode functions correctly

A screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

Works fine.

### Networking tests

This is my first time working with HTTP and networking in general.

#### Ping test

A screenshot of a computer program

Description automatically generated

Where self.setup() is:

A computer screen shot of a program

Description automatically generated

“7124” was chosen, due to it being free on my computer – and it spells the first letters of “GABE” which I thought was kind of cool.

When I run the test, I get the following error:

A screenshot of a computer program

Description automatically generated

This does not spark joy in me – It does not even say where the error is! To tackle this, I placed print statements throughout my code to see where the error is:

A screen shot of a computer error

Description automatically generated

The PING RPC is being sent and received, but no response is being sent – as nothing is being printed after the PING command is sent on the sender side:

A screen shot of a computer program

Description automatically generated

As you can see, “POST Done?” and “Received: ret” should be printed after the POST command, but nothing is printed.

I then put a Try – Except statement around the post() object to make sure of this, (exiting the program after printing “Error on POST.”) and I got the following result:  
A screen shot of a computer

Description automatically generated

The issues were:

* Threading issues in common\_request\_handler: this was fixed by removing the thread used there:

A screen shot of a computer program

Description automatically generated  
*Above: A portion of the fix – old code is commented out, new code can be seen above it.*

Common\_request\_handler was heavily modified, to the following state:

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

### DHT not saving correctly due to pickle

A screenshot of a computer error

Description automatically generated

This is occurring when trying to pickle a DHT object which utilises threading – this occurs when ParallelRouter is the router object.

Python’s built in module “pickle” cannot pickle Thread.lock objects, but the 3rd party extension of pickle “Dill” can, according to [this website](https://itsourcecode.com/typeerror/typeerror-cannot-pickle-_thread-lock-object/).

A screen shot of a computer program

Description automatically generated

I have replaced “pickle” with “dill” in the DHT serialisation methods.



This fixed the issue.

## GUI Tests

A screenshot of a computer

Description automatically generated

* + - * A screenshot of a computer

        Description automatically generatedSettings

This correctly auto fills dht.pickle into default export file.

A screenshot of a computer

Description automatically generated



It has been created successfully in the root folder.

### Could I make it save to somewhere other than the project directory?

A screenshot of a computer

Description automatically generated



Yes!

A screenshot of a computer

Description automatically generated

A black background with white text

Description automatically generated

It also works for forward slashes too.

### Can I make it save to a folder which does not exist?

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

No – this is not fatal, the GUI stays running, it might cause some interesting behaviour though, so I will catch this.

I also need to make some dialogue which clearly states in the GUI if it has been created or not.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

### Our contact not saving correctly

When exporting our contact to a JSON file, it was calling self.id, self.url, self.port and self.protocol\_type. This was both the names of the parameters wanted, and the labels they were being displayed as.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generated

This was fixed by adding “\_label” to all the labels.

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

I will add some GUI dialogue here as well.

A screenshot of a computer

Description automatically generated

### Bootstrapped contact not responding

A screen shot of a computer

Description automatically generated

The node we are attempting to bootstrap from is not responding. I think this is because in my GUI DHT code I don’t start a server. This also does not happen automatically at any other point in my code.

My first thought was to fix this by adding this to the end of MainGUI.initialise\_kademlia:

A screen shot of a computer program

Description automatically generated

This did not work because our\_ip is a global IP, it works when replaced with “127.0.0.1” for our machine.

A screenshot of a computer program

Description automatically generated

There are 2 problems here which require addressing:

* The required tuple does not return from find\_node() on all paths.
* I cannot reach the other node.

Problem 1 was solved easily, by adding an else statement, which I had forgotten previously:

A computer screen shot of a program code

Description automatically generated

Problem 2 is harder; this is because I have not port forwarded the devices the network is running on. Port forwarding fixed the issue.

### A screenshot of a computer Description automatically generatedDownloading

A screenshot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generated

## Unit Test Tables

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ref | Description | Type | Expected | Pass/Fail | Comments or Corrections to be made |
| Test-add-to-contact | Contact added to a full bucket. | Erroneous | “Too Many Contacts Error” should be raised. | Pass |  |
| Test-too-many contacts | Contact added to an almost full bucket. | Boundary | No exceptions should be raised, length of bucket contacts should be Constants.K | Pass | This implies standard behaviour of contacts being added to a relatively empty bucket, as it is required as a prerequisite to this test. |
| Test-no-funny business | Makes sure too empty buckets formed differently create the same outcome | Typical | They should both contain the same contacts (nothing) | Failed now passes | This was fixed in “Problem 1 – Buckets split incorrectly” |
| Test-unique-id add | Adding K contacts to bucket list. | Boundary | Bucket list should not split into separate buckets, and K contacts should exist in one bucket. | Pass |  |
| Test-duplicate-id | Adding 1 contact to a bucket list twice. | Typical | The bucket list should realise that the contact ID already exists in the buckets, therefore it should not be added. | Pass |  |
| Test-bucket-split | Adding K + 1 contacts to an empty bucket list. | Typical | The bucket list should split into 2 separate buckets. | Pass |  |
| Test-force-failed-add | Creates a bucket list composed of K ID’s, with a depth of 5 in the range 2159 to 2160 – 1, along with another Contact with ID in range 0 to 2159 – 1.  Then another contact should be added with ID >= 2159. | Typical | Bucket split should occur, with 1 contact in the first bucket, and 20 contacts in the second bucket.  Then when the 22nd contact is added, nothing should have changed, due to the depth of the bucket it’s being added to MOD 5 is 0. | Pass |  |
| Test-get-close-contacts-ordered | Adds 100 random contacts to a nodes bucket list, then FIND\_NODE is performed. | Typical | K Contacts should be returned; Returned contacts should be ordered by distance.  It should have returned the smallest ID’s possible, as host ID = 0. | Pass |  |
|  |  |  |  |  |  |

## Input validation test tables

I tested this by creating a a new scratch python file, importing my GUI file and launching each individual frame and window in a new customtkinter window.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Window/Frame name | ID # | Type of Data | Pass/Fail | Comments |
| StatusWindow | 1 | Typical | Pass | Passes standard strings to status and copy data, copy data can also be None. |
| StatusWindow | 2 | Boundary | Pass | Passes integers and None to copy data, status message remains as a string. |
| StatusWindow | 3 | Erroneous | Pass | Passes None, type objects and integers to status message. Also passes type objects and integers to copy data. Tkinter is surprisingly robust |
| Settings | 4 | Typical | Pass | Tests settings object when provided with differing DHT objects: These used lots of things varying through VirtualProtocol, VirtualStorage, Router, ParallelRouter, Secondary-JSON-Storage, using the storage-factory mechanism of DHT, and the preset storage mechanisms. DHT is exported successfully each time, when saving to “dht.pickle”. |
| Settings | 5 | Boundary | Pass | Tests settings object when provided with JSONStorage objects that end in suffixes that differ from “JSON”. DHT is exported successfully each time |
| Settings | 6 | Erroneous | Fail – now passes | The DHT cannot This failing will never matter because it is an issue with the DHT class itself. And the user cannot provide an input which is not a string into IP textbox entry. |
| Settings | 7 | Typical | Pass | Saving to file with a suffix – it does not have to be .pickle and it still works. |
| Settings | 8 | Boundary | Pass | Saving to file without a suffix. |
| Settings | 9 | Boundary | Fail – now passes | Saving to “dht/dht” fails. This was fixed by adding a line to DHT.save() saying to make a directory if it did not exist. I assumed python did this automatically when writing to a file, I was wrong. I also tested “C:/dht/dht” which worked. |
| Settings | 10 | Erroneous | Fail – now passes | You cannot save to an empty file or to a file which is a white space. This is to be expected, and an appropriate error is raised. |
| ErrorWindow | 1 | Typical | Pass | Passes standard strings to the error message. |
| ErrorWindow | 2 | Boundary | Pass | Passes integers to the error message. |
| ErrorWindow | 3 | Erroneous | Pass | Passes None and type objects to error message. Also passes type objects and integers to copy data. Tkinter is surprisingly robust.  The reason for IDs being reused is that ErrorWindow and StatusWindow use the same method for displaying data, the only difference is the prebuilt title and the copy\_data message in StatusWindow. |
| MainGUI | N/A | N/A | N/A | This does not allow for any user input, it just calls other frames, which I am testing individually. The only thing of node is that this controls the DHT object. |
| UploadFrame | 11 | Typical |  | Uploads a file in the current working directory, and from external directory. |
| UploadFrame | 12 | Boundary | Pass | Files with no file endings in external directories can be stored, and they can in CWD as well. |
| UploadFrame | 13 | Erroneous | Pass | Files that do not exist are not uploaded. |
| DownloadFrame | 14 | Typical | Pass | Files that exist on a known peer are installed. |
| DownloadFrame | 15 | Erroneous | Pass | Files that we do not know about are not installed. |
| DownloadFrame | 16 | Typical | Pass | Files are searched for when an ID object is given. |
| DownloadFrame | 17 | Erroneous | Pass | Error message is displayed when an ID which is not a number is going – this checking method states that “-1” is not a number, as it is not numeric. But this does not pose an issue because the minimum ID is 0 |
| DownloadFrame | 18 | Boundary | Fail – now passes | 2^160 + 1 is out of range, and decimals within ID range are not accepted, since they are not fully numeric. 0 is also not considered in range. This should be. This was fixed by adding a “greater than or equal to” instead of a standard equals sign in my error checking. 2 ^ 160 is also out of range. |
| MainNetworkFrame | 19 | Typical | Pass | All buttons function as expected – back goes back and “Download” directs to DownloadFrame etc. |
| LoadDHTFromFileFrame | 20 | Typical | Fail – now passes | Attempts to load from absolute and non-absolute file path. The non-absolute file path errored the program due to an error in my directory creation function. |
| LoadDHTFromFileFrame | 21 | Erroneous | Fail – now passes | Refers to file which does not contain valid data. I had not handled an invalid data exception so I add that then it worked fine. |
| JoinNetworkMenuFrame | 22 | Typical | Pass | All buttons worked as expected. There is no input in this screen to be tested. It is purely just 4 buttons which link to different frames. |
| BootstrapFromJSONFrame | 23 | Typical | Fail – now passes | I had forgotten to redirect the bootstrapped peer to the new network screen, nothing had gone wrong. It is fixed. |
| BootstrapFromJSONFrame | 24 | Erroneous | Fail – now passes | I forgot to handle errors raised by the bootstrap. It is fixed now. |
| BootstrapFrame | 25 | Typical | Fail – now passes | I had my regex statement the wrong way round – I was comparing the regex pattern against the string when it should have been vice versa. |
| BootstrapFrame | 27 | Boundary | Fail – now passes | This tests the IDs 2^160 and 2^160 – 1. The first crashed the program by causing an unhandled ValueError, which I then fixed by adding a forgotten greater than or equal to symbol in my code. 2^160 – 1 is in range. |
| BootstrapFrame | 26 | Erroneous | Passes | Appropriate errors are raised when |

## Unit test return

A screenshot of a computer

Description automatically generatedFor unit test code, see Appendix: Unit tests.

# EVALUATION

## Client Feedback

I interviewed Dan for feedback on my solution for him. I first asked him for ratings out of 10 for different elements of my design:

Me: “How would you rate it on ease to use?”

Dan: “6/10, u kind of have to know how kademlia works to use it, but to be fair theres not too much on the screen at once so once you get to the big download and upload buttons it’s a bit easier”

*I would like to add here that he uploaded a file with little to no difficulty, he experienced a bit of trouble with bootstrapping into the network, and he was extremely confused on the 2 different types of keys being used – I had to help him download the file because he forgot to copy the key when he uploaded the file, so he had to manually open the storage file.*

Me: “Would you say its lightweight enough for you”

Dan: “wdym”

Me: “you said you didn’t like OneDrive because it was not lightweight enough when we first chatted about this”

Dan: “yeah its lightweight enough I could probably run it on my raspberry pi”

Me: “bro why didn’t u tell me u had a raspberry pi D:”

Dan: “oops”

Me: “anyway rate lightweightness out of 10”

Dan: “its like a 9 until I go to download a file and the whole program lags until its done so maybe like a 7 idk”

Me: “will you use it instead of onedrive?”

Dan: “maybe sometimes but onedrives v easy to just open and use online and for this ive got to boot the app itself which is effort”

I think he was satisfied but I don’t think its enough to make him switch away from the current solution permanently.

## Scope for further development

This project could be expanded in a variety of ways:

* Download bars could be added when downloading a file, so the user knows how much has been downloaded. This would be quite easy to implement using HTTP request/response streams; you would read the “Content-Length” header and the percentage would be the file size you have received in the stream at that moment divided by the content length header, all multiplied by 100.

## Comparison of achieved solution versus Objectives

### Have a system which can send data between 2 devices using IP and port number.

I have created a TCP server which can handle requests sent by a device, and a TCP protocol which uses the “requests” library to send requests. These use HTTP to communicate. HTTP is insecure (inherently less secure than HTTPS, given the names), but the library “http.server” I was using to implement the server had no mention of HTTPS, and I could not find an easy implementation of it in Python. This would have easily have allowed for secure communication between devices.

The system I have created uses a mix of IP, port and Kademlia ID. We only have to know one contacts ID, port and Kademlia ID and then we can bootstrap and inherit the contacts known by our known peer. This also allows us to indirectly contact an IP and port that we don’t know, by sending a Kademlia FIND\_VALUE RPC to attempt to find a key-value pair stored by an unknown peer on the network. This works by contacting one of the peers we do know, and then they attempt to find the value, if they cannot, they will return “K” (20) contacts that are closer than them to the key in the key-value pair we are looking for. We will then send the FIND\_VALUE RPC to “ALPHA” (20 or 3, depending on where you look, I chose to use 20 for the implementation due to its larger breadth) of them. They will then perform the same thing. Each node does not know just a random set of other nodes though, each node maintains a bucket list containing at most 160 buckets, each containing K contacts in a given range, which should be powers of 2 at either end.

### Use some method to determine which device has the requested data

When I chose to use Kademlia I said it should do the following things:

* + Must have a functioning implementation of “KBuckets”, these are the lists of ID’s that each device knows about.
    - Each must split properly and at the correct time.
    - Each KBucket should have a range with high and low boundaries on powers of 2.
    - Contacts are added to the correct KBucket from the bucket list.
  + Create server to handle incoming requests.
  + Unresponsive nodes should be evicted from the network.
  + Method to lookup nodes on the network.
  + Method to find nodes on the network.
  + (Optional) ID object which can be used interchangeably with integers.
  + Ability to send RPC (Remote Procedure Call) requests to other nodes on the network

All of these have been created, I even have 2 servers to handle requests (both use threading), but one uses subnets and is for debugging.

### Query devices asynchronously.

I created a router object which can perform RPC calls in parallel with one another, using the “threading” library. This used a semaphore to coordinate the threads so they would not hog my computers memory. I would say the parallelism was a success, because it cut down lookup times drastically, because the program didn’t have to wait for lots of nodes to respond. I also replaced my original idea of using “HTTPServer” with the “HTTPThreadingServer” class, which allowed for multiple requests to be handled at the same time, also using threads. This fixed the issue of serially addressing nodes, because some nodes will have to handle multiple requests at once.

### GUI to coordinate the transfer process

I created a GUI which uses window and frame classes to direct the user around the network, making sure they can only access buttons which they are able to use. For example, the settings icon is hidden before Kademlia is initialised, because the only things it allowed the user to do was export the Kademlia instance, and their contact. Neither of these are possible before Kademlia is initialised.

### Must be scalable and lightweight.

Since Kademlia was used, and not a simpler solution, it allows for extreme scalability. Kademlia will perform even more successfully the more nodes there are. This does create the downside that the network ma

### Should be aesthetically pleasing.

I have used “customtkinter” instead of normal tkinter to achieve this. I could have created a web app in retrospect and it would have saved me a lot of time – I am familiar with the Laravel framework for PHP so that could have been used instead. “customtkinter” has been a headache for me. Even though it is just a prettier version of tkinter, it has some very bad error handling, which makes debugging it a pain in the neck.

# APPENDIX

## Code

### Buckets.py

*from* datetime *import* datetime  
*from* os.path *import* commonprefix  
  
*from* kademlia.constants *import* Constants  
*from* kademlia.contact *import* Contact  
*from* kademlia.errors *import* BucketDoesNotContainContactToEvictError, OurNodeCannotBeAContactError, OutOfRangeError, \  
 RPCError, TooManyContactsError  
*from* kademlia.id *import* ID  
  
  
*class* KBucket:  
  
 *def* \_\_init\_\_(*self*,  
 initial\_contacts: list[Contact] | *None* = *None*,  
 low: int = 0,  
 high: int = 2 \*\* 160):  
 *"""  
 Initialises a k-bucket with a specific ID range,  
 initially from 0 to 2\*\*160.  
 """  
 if* initial\_contacts *is None*: *# Fix for instead of setting initial\_contacts = []* initial\_contacts = []  
  
 *self*.contacts: list[Contact] = initial\_contacts  
 *self*.\_low: int = low  
 *self*.\_high: int = high  
 *self*.time\_stamp: datetime = datetime.now()  
 *# self.lock = WithLock(Lock())  
  
 def* low(*self*) -> int:  
 *return self*.\_low  
  
 *def* high(*self*) -> int:  
 *return self*.\_high  
  
 *def* is\_full(*self*) -> bool:  
 *"""  
 This INCLUDES K, so if there are 20 inside, no more can be added.  
 :return: Boolean saying if it's full.  
 """  
 return* len(*self*.contacts) >= Constants.K  
  
 *def* contains(*self*, id: ID) -> bool:  
 *"""  
 Returns boolean determining whether a given contact ID is in the k-bucket.  
 """  
  
 # replaceable  
 return* any(id == contact.id *for* contact *in self*.contacts)  
  
 *def* touch(*self*) -> *None*:  
 *self*.time\_stamp = datetime.now()  
  
 *def* is\_in\_range(*self*, other\_id: ID) -> bool:  
 *"""  
 Determines if a given ID is within the range of the k-bucket.  
 :param other\_id: The ID to be checked.  
 :return: Boolean saying if it's in the range of the k-bucket.  
 """  
 return self*.\_low <= other\_id.value <= *self*.\_high  
  
 *def* add\_contact(*self*, contact: Contact) -> *None*:  
 *if self*.is\_full():  
 *raise* TooManyContactsError(  
 f"KBucket is full - (length is {len(*self*.contacts)}).")  
 *elif not self*.is\_in\_range(contact.id):  
 *raise* OutOfRangeError("Contact ID is out of range.")  
 *elif* contact *not in self*.contacts:  
 *self*.contacts.append(contact)  
 *else*:  
 print("[Client] Contact already in KBucket.")  
  
 *def* depth(*self*) -> int:  
 *"""  
 "The depth is just the length of the prefix shared by all nodes in  
 the k-bucket’s range." Do not confuse that with this statement in the  
 spec: “Define the depth, h, of a node to be 160 - i, where i is the  
 smallest index of a nonempty bucket.” The former is referring to the  
 depth of a k-bucket, the latter the depth of the node.  
 """  
  
 return* len(*self*.shared\_bits())  
  
 *def* shared\_bits(*self*) -> str:  
 *"""  
 Return the longest shared binary prefix between all  
 contacts in the kbucket. This does not "0b" before the binary.  
 """* shared\_bits = commonprefix([i.id.bin() *for* i *in self*.contacts])  
 *# print("shared bits", shared\_bits)  
 return* shared\_bits  
  
 *def* split(*self*) -> tuple:  
 *"""  
 Splits KBucket in half, returns tuple of type (KBucket, KBucket).  
 """  
 # This doesn't work when all contacts are bunched towards one side of the KBucket.  
 # It's in the spec so I'm keeping it, it also means it stays nice and neat* midpoint: int = (*self*.\_low + *self*.\_high) // 2 *# This will always be an integer, but // is faster than /.  
  
 # Gets the median of all contacts inside the KBucket (rounding up in even # of contacts)  
 # contact\_ids\_asc = sorted([c.id.value for c in self.contacts])  
 # median\_of\_contact\_id: int = median\_high(contact\_ids\_asc)  
 # midpoint = median\_of\_contact\_id* k1: KBucket = KBucket(low=*self*.\_low, high=midpoint)  
 k2: KBucket = KBucket(low=midpoint, high=*self*.\_high)  
 *for* c *in self*.contacts:  
 *if* c.id.value < midpoint:  
 k1.add\_contact(c)  
 *else*:  
 k2.add\_contact(c)  
  
 *return* k1, k2  
  
 *def* replace\_contact(*self*, contact: Contact) -> *None*:  
 *"""replaces contact, then touches it"""* contact\_ids = [c.id *for* c *in self*.contacts]  
 index = contact\_ids.index(contact.id)  
 contact.touch()  
 *self*.contacts[index] = contact  
  
 *def* evict\_contact(*self*, contact: Contact) -> *None*:  
 *if self*.contains(contact.id):  
 *self*.contacts.remove(contact)  
 *else*:  
 *raise* BucketDoesNotContainContactToEvictError(  
 "Contact not found."  
 )  
  
  
*class* BucketList:  
  
 *def* \_\_init\_\_(*self*, our\_contact: Contact):  
 *"""  
 :param our\_contact: Our contact  
 """  
 self*.dht = *None  
 self*.buckets: list[KBucket] = [KBucket()]  
 *# first k-bucket has max range  
 self*.our\_id: ID = our\_contact.id  
 *self*.our\_contact: Contact = our\_contact  
  
 *# create locking object  
 # self.lock = WithLock(Lock())  
  
 def* can\_split(*self*, kbucket: KBucket) -> bool:  
 *# kbucket.HasInRange(ourID) || ((kbucket.Depth() % Constants.B) != 0)  
 """  
 The depth to which the bucket has split is based on the number of bits  
 shared in the prefix of the contacts in the bucket. With random IDs,  
 this number will initially be small, but as bucket ranges become more  
 narrow from subsequent splits, more contacts will begin the share the  
 same prefix and the bucket when split, will result in less “room” for  
 new contacts. Eventually, when the bucket range becomes narrow enough,  
 the number of bits shared in the prefix of the contacts in the bucket  
 reaches the threshold b, which the spec says should be 5.  
 """  
 # with self.lock:  
 # TODO: What is self.node?  
  
 return* (kbucket.is\_in\_range(*self*.our\_id)  
 *or* (kbucket.depth() % Constants.B != 0))  
  
 *def* \_get\_kbucket\_index(*self*, other\_id: ID) -> int:  
 *"""  
 Returns the first k-buckets index in the bucket list  
 which has a given ID in range. Returns -1 if not found.  
 """  
  
 # with self.lock:  
 for* i *in* range(len(*self*.buckets)):  
 *if self*.buckets[i].is\_in\_range(other\_id):  
 *return* i  
 *return* -1  
  
 *def* get\_kbucket(*self*, other\_id: ID) -> KBucket:  
 *"""  
 Returns the first k-bucket in the bucket list  
 which has a given ID in range. Raises an error if none are found  
 - this should never happen!  
 :param other\_id: ID to used to determine range.  
 :return: the first k-bucket which is in range.  
 """  
  
 try*:  
 bucket = *self*.buckets[*self*.\_get\_kbucket\_index(other\_id)]  
 *return* bucket  
  
 *except* IndexError:  
 *raise* OutOfRangeError(f"ID: {id} is not in range of bucket-list.")  
  
 *def* add\_contact(*self*, contact: Contact) -> *None*:  
 *"""  
 Adds a contact to a k-bucket in the list, this is determined by the range of k-buckets in the lists.  
 This range should span the entire ID space - so there should always be a k-bucket to be added.  
  
 If we can't add it, it will be added to DHT pending contacts.  
  
 This raises an error if we try to add ourselves to the k-bucket.  
  
 :param contact: Contact to be added, this is touched in the process.  
 :return: None  
 """  
 if self*.our\_id == contact.id:  
 *raise* OurNodeCannotBeAContactError(  
 "Cannot add ourselves as a contact.")  
  
 contact.touch() *# Update the time last seen to now  
  
 # print("[Client] Add contact called.")  
 # with self.lock:* kbucket: KBucket = *self*.get\_kbucket(contact.id)  
 *if* kbucket.contains(contact.id):  
 print("[Client] Contact already in KBucket.")  
 *# replace contact, then touch it* kbucket.replace\_contact(contact)  
 *elif* kbucket.is\_full():  
 print("[Client] Kbucket is full.")  
 *if self*.can\_split(kbucket):  
 print("[Client] Splitting!")  
 *# Split then try again* k1, k2 = kbucket.split()  
 *# print(f"K1: {len(k1.contacts)}, K2: {len(k2.contacts)}, Buckets: {self.buckets}")* index: int = *self*.\_get\_kbucket\_index(contact.id)  
  
 *# adds the two buckets to 2 separate buckets.  
 self*.buckets[index] = k1 *# Replaces original KBucket  
 self*.buckets.insert(index + 1, k2) *# Adds a new one after it  
 # print(self.buckets)  
 self*.add\_contact(  
 contact  
 ) *# Unless k <= 0, This should never cause a recursive loop  
 else*:  
 print("[Client] Cannot split")  
 last\_seen\_contact: Contact = sorted(  
 kbucket.contacts, key=*lambda* c: c.last\_seen)[0]  
 error: RPCError | *None* = last\_seen\_contact.protocol.ping(  
 *self*.our\_contact)  
 *if* error:  
 *# Unresponsive* print("[Client] Node is unresponsive")  
 *if self*.dht: *# tests may not initialise a DHT* print("[Client] Delaying eviction")  
 *self*.dht.delay\_eviction(last\_seen\_contact, contact)  
 *else*:  
 *# still can't add the contact ,so put it into the pending list* print("[Client] Node is responsive.")  
 *if self*.dht:  
 print("[Client] Adding node to DHT pending...")  
 *self*.dht.add\_to\_pending(contact)  
  
 *else*:  
 *# Bucket is not full, nothing special happens.  
 # print("[Client] Adding contact to bucket")* kbucket.add\_contact(contact)  
  
 *def* get\_close\_contacts(*self*, key: ID, exclude: ID) -> list[Contact]:  
 *"""  
 Brute force distance lookup of all known contacts, sorted by distance.  
 Then we take K of the closest.  
 :param key: The ID for which we want to find close contacts.  
 :param exclude: The ID to exclude (the requesters ID).  
 :return: List of K contacts sorted by distance.  
 """  
 # print(key, exclude)  
 # with self.lock:* contacts = []  
 *# print(self.buckets)  
 for* bucket *in self*.buckets:  
 *# print(bucket.contacts)  
 for* contact *in* bucket.contacts:  
 *# print(contact.id.value)  
 # print(f"Exclude: {exclude}")  
  
 if* contact.id != exclude:  
 contacts.append(contact)  
 *# print(contacts)* contacts = sorted(contacts, key=*lambda* c: c.id ^ key)[:Constants.K]  
 *if* len(contacts) > Constants.K *and* Constants.DEBUG:  
 *raise* ValueError(  
 f"Contacts should be smaller than or equal to K. Has length {len(contacts)}, "  
 f"which is {Constants.K - len(contacts)} too big.")  
 *return* contacts  
  
 *def* contacts(*self*) -> list[Contact]:  
 *"""  
 Returns a list of all contacts in the bucket list.  
 :return: All contacts in the bucket list.  
 """* contacts = []  
 *for* bucket *in self*.buckets:  
 *for* contact *in* bucket.contacts:  
 contacts.append(contact)  
 *return* contacts  
  
 *def* contact\_exists(*self*, contact: Contact) -> bool:  
 *return* contact *in self*.contacts()

### Constants.py

*from* dataclasses *import* dataclass  
  
  
@dataclass  
*class* Constants:  
 K = 20  
 B = 5 *# or 160 according to https://xlattice.sourceforge.net/components/protocol/kademlia/specs.html* REQUEST\_TIMEOUT = 0.5 *# 500ms* ID\_LENGTH\_BYTES = 20  
 ID\_LENGTH\_BITS = 160  
 MAX\_THREADS = 20  
 RESPONSE\_WAIT\_TIME = 10 *# in ms* BUCKET\_REFRESH\_INTERVAL = 60 \* 60 \* 1000 *# hourly in ms* KEY\_VALUE\_REPUBLISH\_INTERVAL = 60 \* 60 \* 1000 *# hourly in ms* KEY\_VALUE\_EXPIRE\_INTERVAL = 60 \* 60 \* 1000 *# hourly in ms* ORIGINATOR\_REPUBLISH\_INTERVAL = 24 \* 60 \* 60 \* 1000 *# every 24 hours in ms* EXPIRATION\_TIME\_SEC = 24 \* 60 \* 60 *# every 24 hours in seconds* EVICTION\_LIMIT = 5  
 MAX\_PORT\_RETRIES = 100  
 TRY\_CLOSEST\_BUCKET = *True* DEBUG = *True* DHT\_SERIALISED\_SUFFIX = "dht"  
  
 *if* DEBUG:  
 A: int = 3  
 *else*:  
 A: int = 20

### Contact.py

*from* datetime *import* datetime  
*from* typing *import* Optional  
  
*from* kademlia.id *import* ID  
*from* kademlia.interfaces *import* IProtocol  
*from* kademlia.constants *import* Constants  
  
  
*class* Contact:  
  
 *def* \_\_init\_\_(*self*, id: ID, protocol=*None*):  
 *if* protocol *is None and not* Constants.DEBUG:  
 *raise* ValueError("No protocol given to Contact.")  
 *self*.protocol: Optional[IProtocol] = protocol  
 *self*.id = id  
 *self*.last\_seen: datetime = datetime.now()  
  
 *def* touch(*self*) -> *None*:  
 *"""Updates the last time the contact was seen."""  
 self*.last\_seen = datetime.now()

### dht.py

*import* threading  
*from* datetime *import* datetime, timedelta  
*from* typing *import* Callable, Optional  
  
*import* dill  
  
*from* kademlia *import* helpers  
*from* kademlia.buckets *import* KBucket  
*from* kademlia.constants *import* Constants  
*from* kademlia.contact *import* Contact  
*from* kademlia.dictionaries *import* FindResult  
*from* kademlia.errors *import* BucketDoesNotContainContactToEvictError, RPCError  
*from* kademlia.id *import* ID  
*from* kademlia.interfaces *import* IProtocol, IStorage  
*from* kademlia.node *import* Node  
*from* kademlia.routers *import* BaseRouter  
  
  
*class* DHT:  
 *"""  
 This is the main entry point for our peer to interact with other peers.  
  
 This has multiple purposes:  
 - One is to propagate key-values to other close peers on the network using a lookup algorithm.  
 - Another is to use the same lookup algorithm to search for other close nodes that might have a value that we don’t have.  
 - It is also used for bootstrapping our peer into a pre-existing network.  
  
 """  
  
 def* \_\_init\_\_(*self*,  
 id: ID,  
 protocol: IProtocol,  
 router: BaseRouter,  
 storage\_factory: Callable[[], IStorage] | *None* = *None*,  
 originator\_storage: IStorage | *None* = *None*,  
 republish\_storage: IStorage | *None* = *None*,  
 cache\_storage: IStorage | *None* = *None*):  
 *"""  
  
 We use a wrapper Dht class, which will become the main entry point for our peer,  
 for interacting with other peers. The purposes of this class are:  
  
 - When storing a value, use the lookup algorithm to find other closer peers to  
 propagate the key-value.  
 - When looking up a value, if our peer doesn’t have the value, we again use the  
 lookup algorithm to find other closer nodes that might have the value.  
 - A bootstrapping method that registers our peer with another peer and  
 initializes our bucket list with that peer’s closest contacts.  
  
 Supports different concrete storage types.  
 For example, you may want the cache\_storage to be an in-memory store,  
 the originator\_storage to be a SQL database, and the republish store to be a  
 key-value database.  
  
 :param id: ID associated with the DHT.  
  
 :param protocol: Protocol implemented by the DHT.  
  
 :param storage\_factory: Storage to be used for all storage mechanisms -  
 if specific mechanisms are not provided.  
  
 :param originator\_storage: Pre-existing storage object to be used for main  
 storage.  
  
 :param republish\_storage: This contains key-values that have been republished  
 by other peers.  
  
 :param cache\_storage: Short term storage.  
  
 :param router: Router object associated with the DHT.  
 """  
  
 if* originator\_storage:  
 *self*.\_originator\_storage = originator\_storage  
 *elif* storage\_factory:  
 *# if storage\_factory == SecondaryJSONStorage:  
 # self.\_originator\_storage = storage\_factory(  
 # filename=f"{id.value}/originator\_storage.json")  
 # else:  
 self*.\_originator\_storage = storage\_factory()  
 *else*:  
 *raise* TypeError(  
 "Originator storage must take parameter originator\_storage,"  
 " or be generated by generated by parameter storage\_factory.")  
  
 *if* republish\_storage:  
 *self*.\_republish\_storage = republish\_storage  
 *elif* storage\_factory:  
 *# if storage\_factory == SecondaryJSONStorage:  
 # self.\_republish\_storage = storage\_factory(  
 # filename=f"{id.value}/republish\_storage.json")  
 # else:  
 self*.\_republish\_storage = storage\_factory()  
 *else*:  
 *raise* TypeError(  
 "Republish storage must take parameter republish\_storage,"  
 " or be generated by generated by parameter storage\_factory.")  
  
 *if* cache\_storage:  
 *self*.\_cache\_storage = cache\_storage  
 *elif* storage\_factory:  
 *# if storage\_factory == SecondaryJSONStorage:  
 # self.\_cache\_storage = storage\_factory(  
 # filename=f"{id.value}/cache\_storage.json")  
 # else:  
 self*.\_cache\_storage = storage\_factory()  
 *else*:  
 *raise* TypeError(  
 "Cache storage must take parameter cache\_storage,"  
 " or be generated by generated by parameter storage\_factory.")  
  
 *self*.pending\_contacts: list[Contact] = []  
 *self*.our\_id = id  
 *self*.our\_contact = Contact(id=id, protocol=protocol)  
 *# if router.node:  
 # self.node: Node = router.node  
 self*.node: Node = Node(*self*.our\_contact,  
 storage=*self*.\_republish\_storage,  
 cache\_storage=*self*.\_cache\_storage)  
 *self*.node.dht = *self  
 self*.node.bucket\_list.dht = *self  
 self*.\_protocol = protocol  
 *self*.\_router: BaseRouter = router  
 *self*.\_router.node = *self*.node  
 *self*.\_router.dht = *self  
 self*.eviction\_count: dict[int, int] = {}  
  
 *def* router(*self*) -> BaseRouter:  
 *return self*.\_router  
  
 *def* protocol(*self*) -> IProtocol:  
 *return self*.\_protocol  
  
 *def* originator\_storage(*self*) -> IStorage:  
 *return self*.\_originator\_storage  
  
 *def* store(*self*, key: ID, val: str) -> *None*:  
 print(f"[Client] Storing value at {key}.")  
 *self*.touch\_bucket\_with\_key(key)  
 *# We're storing to K closer contacts  
 self*.\_originator\_storage.set(key, val)  
 *self*.store\_on\_closer\_contacts(key, val)  
  
 *def* find\_value(*self*, key: ID) -> tuple[bool, list[Contact] | *None*, str | *None*]:  
 *"""  
 Attempts to find a given value.  
 First it checks our originator storage. If the given key does not have a value in our storage,  
 it will use Router.lookup() to attempt to find it. If there is no value found from router.lookup(), the value  
 returned will be None.  
 If there is a value found from router.lookup(), the value will be stored on the closest contact to us, if  
 one exists.  
 :param key: Key to search for value pair.  
 :return: Found: bool (If it is found or not), contacts: list[Contact], val: str | None (value returned)  
 """* print("touch bucket with key")  
 *self*.touch\_bucket\_with\_key(key)  
 contacts\_queried: list[Contact] = []  
  
 *# ret (found: False, contacts: None, val: None)* contacts: list[Contact] | *None* = *None  
 # - Add to docstring when finished* val: str | *None* = *None* found, our\_val = *self*.\_originator\_storage.try\_get\_value(key)  
 *# There has to be a better way to do this.  
 if* our\_val:  
 found = *True* val = our\_val  
 *else*:  
 found, our\_val = *self*.\_republish\_storage.try\_get\_value(key)  
 *if* our\_val:  
 found = *True* val = our\_val  
 *else*:  
 found, our\_val = *self*.\_cache\_storage.try\_get\_value(key)  
 *if* our\_val:  
 found = *True* val = our\_val  
 *else*:  
 lookup: FindResult = *self*.\_router.lookup(  
 key, *self*.\_router.rpc\_find\_value)  
 *if* lookup["found"]:  
 found = *True* contacts = *None* val = lookup["val"]  
 *# Find the closest contact (other than the one the value was found by)  
 # in which to "cache" the key-value.* store\_to: Contact | *None* = *None  
 for* c *in* lookup["contacts"]:  
 *if* c.id.value != lookup["found\_by"].id.value:  
 store\_to: Contact | *None* = c  
 *break  
  
 if* store\_to:  
 separating\_nodes: int = *self*.\_get\_separating\_nodes\_count(*self*.our\_contact, store\_to)  
 print("Separating nodes:", separating\_nodes) *# TODO: remove* exp\_time\_sec: int = Constants.EXPIRATION\_TIME\_SEC // (2 \*\* separating\_nodes)  
 error: RPCError = store\_to.protocol.store(*self*.node.our\_contact, key, lookup["val"],  
 exp\_time\_sec=exp\_time\_sec)  
 *self*.handle\_error(error, store\_to)  
  
 *return* found, contacts, val  
  
 *def* touch\_bucket\_with\_key(*self*, key: ID) -> *None*:  
 *"""  
 Touches a KBucket with a given key from the bucket list.  
 :return: Returns nothing.  
 """  
 self*.node.bucket\_list.get\_kbucket(key).touch()  
  
 *def* store\_on\_closer\_contacts(*self*, key: ID, val: str) -> *None*:  
 now: datetime = datetime.now()  
 kbucket: KBucket = *self*.node.bucket\_list.get\_kbucket(key)  
 contacts: list[Contact]  
 *if* (now - kbucket.time\_stamp) < timedelta(  
 milliseconds=Constants.BUCKET\_REFRESH\_INTERVAL):  
 *# Bucket has been refreshed recently, so don't do a lookup as we  
 # have the k closest contacts.* contacts: list[Contact] = *self*.node.bucket\_list.get\_close\_contacts(  
 key=key, exclude=*self*.node.our\_contact.id)  
 *else*:  
 contacts: list[Contact] = *self*.\_router.lookup(  
 key, *self*.\_router.rpc\_find\_nodes)["contacts"]  
  
 *for* c *in* contacts:  
 error: RPCError | *None* = c.protocol.store(  
 sender=*self*.node.our\_contact, key=key, val=val)  
 *self*.handle\_error(error, c)  
  
 *def* bootstrap(*self*, known\_peer: Contact) -> *None*:  
 *"""  
 This is how we join the network.  
  
 We bootstrap our peer by contacting a known peer in the network, adding its contacts  
 to our list, then getting the contacts for other peers not in the  
 bucket range of our known peer we're joining.  
 :param known\_peer: Peer we know / are bootstrapping from.  
 :return: None  
 """* print("[Client] Bootstrapping from known peer.")  
 *# print(f"Adding known peer with ID {known\_peer.id}")  
 self*.node.bucket\_list.add\_contact(known\_peer)  
  
 *# UNITTEST NOTES: This should return something in test\_bootstrap\_outside\_bootstrapping\_bucket,  
 # it isn't at the moment.  
 # find\_node() should return the bucket list with the contact who knows 10 other contacts  
 # it does.  
  
 # finds K close contacts to self.our\_id, excluding self.our\_contact* contacts, error = known\_peer.protocol.find\_node(  
 sender=*self*.our\_contact, key=*self*.our\_id)  
 *self*.handle\_error(error, known\_peer)  
 *if not* error.has\_error():  
 *# print("NO ERROR")  
  
 # add all contacts the known peer DIRECTLY knows  
 for* contact *in* contacts:  
 *self*.node.bucket\_list.add\_contact(contact)  
  
 known\_peers\_bucket: KBucket = *self*.node.bucket\_list.get\_kbucket(  
 known\_peer.id)  
  
 *# Resolve the list now, so we don't include additional contacts  
 # as we add to our bucket additional contacts.* other\_buckets: list[KBucket] = [  
 i *for* i *in self*.node.bucket\_list.buckets  
 *if* i != known\_peers\_bucket  
 ]  
 *for* other\_bucket *in* other\_buckets:  
 *self*.\_refresh\_bucket(  
 other\_bucket  
 ) *# UNITTEST Notes: one of these should contain the correct contact  
 else*:  
 *raise* error  
  
 *def* \_refresh\_bucket(*self*, bucket: KBucket) -> *None*:  
 *"""  
 Refreshes the given Kademlia KBucket by updating its last-touch timestamp,  
 obtaining a random ID within the bucket's range, and attempting to find  
 nodes in the network with that random ID.  
  
 The method touches the bucket to update its last-touch timestamp, generates  
 a random ID within the bucket's range, and queries nodes in the network  
 using the Kademlia protocol to find nodes with the generated ID. If successful,  
 the discovered contacts are added to the Kademlia node's bucket list.  
  
 Note:  
 The contacts collection for the given bucket might change during the operation,  
 so it is isolated in a separate list before iterating over it.  
  
 :param bucket: The KBucket to be refreshed.  
 :returns: Nothing.  
 """* bucket.touch()  
 random\_id: ID = ID.random\_id\_within\_bucket\_range(bucket)  
  
 *# put in a separate list as contacts collection for this bucket might change.* contacts: list[Contact] = bucket.contacts  
 *for* contact *in* contacts:  
 *# print(contact.id, contact.protocol.node.bucket\_list.contacts())* new\_contacts, timeout\_error = contact.protocol.find\_node(  
 *self*.our\_contact, random\_id)  
 *# print(contacts.index(contact) + 1, "new contacts", new\_contacts)  
 self*.handle\_error(timeout\_error, contact)  
 *if* new\_contacts:  
 *for* other\_contact *in* new\_contacts:  
 *self*.node.bucket\_list.add\_contact(other\_contact)  
  
 *def* \_setup\_bucket\_refresh\_timer(*self*) -> *None*:  
 *"""  
 Sets up the refresh timer to re-ping KBuckets.  
  
 From the spec:  
 “Buckets are generally kept fresh by the traffic of requests traveling through nodes. To handle pathological  
 cases in which there are no lookups for a particular ID range, each node refreshes any bucket to which it has  
 not performed a node lookup in the past hour. Refreshing means picking a random ID in the bucket’s range and  
 performing a node search for that ID.”  
 """* bucket\_refresh\_timer = threading.Timer(Constants.BUCKET\_REFRESH\_INTERVAL / 1000, *self*.\_refresh\_bucket)  
 bucket\_refresh\_timer.auto\_reset = *True* bucket\_refresh\_timer.elapsed += *self*.bucket\_refresh\_timer\_elapsed  
 bucket\_refresh\_timer.start()  
  
 *def* \_bucket\_refresh\_timer\_elapsed(*self*):  
 now: datetime = datetime.now()  
 *# Put into a separate list as bucket collections may be modified.* current\_buckets: list[KBucket] = [  
 b *for* b *in self*.node.bucket\_list.buckets  
 *if* (now - b.time\_stamp) >= timedelta(milliseconds=Constants.BUCKET\_REFRESH\_INTERVAL)  
 ]  
  
 *for* b *in* current\_buckets:  
 *self*.\_refresh\_bucket(b)  
  
 *def* \_key\_value\_republish\_elapsed(*self*) -> *None*:  
 *"""  
 Replicate key values if the key value hasn't been touched within  
 the republish interval. Also don't do a FindNode lookup if the  
 bucket containing the key has been refresed within the refresh  
 interval.  
 """* now: datetime = datetime.now()  
  
 rep\_keys = [  
 k *for* k *in self*.\_republish\_storage.get\_keys()  
 *if* now - *self*.\_republish\_storage.get\_timestamp(k) >=  
 Constants.KEY\_VALUE\_REPUBLISH\_INTERVAL  
 ]  
  
 *for* k *in* rep\_keys:  
 key: ID = ID(k)  
 *self*.store\_on\_closer\_contacts(key,  
 *self*.\_republish\_storage.get(key))  
 *self*.\_republish\_storage.touch(k)  
  
 *def* \_expire\_keys\_elapsed(*self*) -> *None*:  
 *"""  
 Removes expired key-values from republish and cache storage.  
 """  
 self*.\_remove\_expired\_data(*self*.\_cache\_storage)  
 *self*.\_remove\_expired\_data(*self*.\_republish\_storage)  
  
 @staticmethod  
 *def* \_remove\_expired\_data(store: IStorage) -> *None*:  
 now: datetime = datetime.now()  
 *# to list so our key list is resolved now as we remove keys* expired: list[int] = [  
 key *for* key *in* store.get\_keys()  
 *if* (now - store.get\_timestamp(key)) >= timedelta(  
 seconds=store.get\_expiration\_time\_sec(key))  
 ]  
  
 *# expired is a list of all expired keys in the given storage.  
 for* key *in* expired:  
 store.remove(key)  
  
 *def* \_originator\_republish\_elapsed(*self*) -> *None*:  
 *"""  
 Redistributes expired key-value pars if we are the publisher.  
  
  
 Spec: “For Kademlia’s current application (file sharing),  
 we also require the original publisher of a (key,value)  
 pair to republish it every 24 hours. Otherwise, (key,value)  
 pairs expire 24 hours after publication, to limit stale  
 index information in the system. For other applications, such  
 as digital certificates or cryptographic hash to value mappings,  
 longer expiration times may be appropriate.”  
 """* now: datetime = datetime.now()  
  
 keys\_pending\_republish = [  
 key *for* key *in self*.\_originator\_storage.get\_keys()  
 *if* (now -  
 *self*.\_originator\_storage.get\_timestamp(key.value)) >= timedelta(  
 milliseconds=Constants.ORIGINATOR\_REPUBLISH\_INTERVAL)  
 ]  
  
 *for* k *in* keys\_pending\_republish:  
 key: ID = k  
 *# Just use close contacts, don't do a lookup* contacts = *self*.node.bucket\_list.get\_close\_contacts(  
 key, *self*.node.our\_contact.id)  
  
 *for* c *in* contacts:  
 error: RPCError | *None* = c.protocol.store(  
 sender=*self*.our\_contact,  
 key=key,  
 val=*self*.\_originator\_storage.get(key)  
 )  
 *self*.handle\_error(error, c)  
  
 *self*.\_originator\_storage.touch(k.value)  
  
 *def* \_get\_separating\_nodes\_count(*self*, contact\_a: Contact, contact\_b: Contact) -> int:  
 *"""  
 Returns the number of contacts between 2 contacts in our bucket list.  
 :param contact\_a:  
 :param contact\_b:  
 :return:  
 """  
 # get all the contacts, ordered by ID* all\_contacts: list[Contact] = sorted(*self*.node.bucket\_list.contacts(), key=*lambda* c: c.id.value)  
 index\_a = helpers.get\_closest\_number\_index([i.id.value *for* i *in* all\_contacts], contact\_a.id.value)  
 index\_b = helpers.get\_closest\_number\_index([i.id.value *for* i *in* all\_contacts], contact\_b.id.value)  
 count = abs(index\_a - index\_b)  
 *return* count  
  
 *def* handle\_error(*self*, error: RPCError | *None*, contact: Contact) -> *None*:  
 *"""  
 Put the timed out contact into a collection and increment the number  
 of times it has timed out.  
  
 If it has timed out a certain amount, remove it from the bucket  
 and replace it with the most recent pending contact that are  
 queued for that bucket.  
 """  
 if* error:  
 *if* error.has\_error():  
 count = *self*.\_add\_contact\_to\_evict(contact.id.value)  
 *if* count >= Constants.EVICTION\_LIMIT:  
 *self*.\_replace\_contact(contact)  
  
 *def* delay\_eviction(*self*,  
 to\_evict: Contact,  
 to\_replace: Contact) -> *None*:  
 *"""  
 The contact that did not respond (or had an error) gets "n"  
 tries before being evicted and replaced with the most recently  
 seen contact that wants to got into the non-responding contact's  
 K-Bucket  
  
 :param to\_evict: The contact that didn't respond.  
 :param to\_replace: The contact that can replace the  
 non-responding contact.  
 """  
 # Non-concurrent list needs locking  
 # lock(pending\_contacts)  
 # add only if its a new pending contact.  
 if* to\_replace.id *not in* [c.id *for* c *in self*.pending\_contacts]:  
 *self*.pending\_contacts.append(to\_replace)  
  
 key: int = to\_evict.id.value  
 count = *self*.\_add\_contact\_to\_evict(key)  
 *# if the eviction attempts on key reach the eviction limit  
 if* count == Constants.EVICTION\_LIMIT:  
 *self*.\_replace\_contact(to\_evict)  
  
 *def* \_add\_contact\_to\_evict(*self*, key\_to\_evict: int) -> int:  
 *"""  
 Increments how many times we have tried to evict a given key, returning number of attempts.  
 :param key\_to\_evict: to\_evict  
 :return: number of attempts  
 """  
 # self.eviction\_count is a dictionary of ID keys ->  
 # how many times they have been considered for eviction.  
 if* key\_to\_evict *not in self*.eviction\_count:  
 *self*.eviction\_count[key\_to\_evict] = 0  
 *self*.eviction\_count[key\_to\_evict] += 1  
  
 *return self*.eviction\_count[key\_to\_evict]  
  
 *def* \_replace\_contact(*self*, to\_evict: Contact) -> *None*:  
 *"""  
 Replaces an evicted contact with a pending one.  
 :param to\_evict:  
 :return:  
 """* bucket = *self*.node.bucket\_list.get\_kbucket(to\_evict.id)  
 *# Prevent other threads from manipulating the bucket list or buckets  
 # lock(self.node.bucket\_list)  
 self*.\_evict\_contact(bucket, to\_evict)  
 *self*.\_replace\_with\_pending\_contact(bucket)  
  
 *def* \_evict\_contact(*self*, bucket: KBucket, to\_evict: Contact) -> *None*:  
 *"""  
 Removes all attempts to evict to\_evict, then removes it from the given bucket,  
 raising an error if it is not in the bucket.  
 :param bucket:  
 :param to\_evict:  
 :return:  
 """* print("[Client] Evicting contact from bucket.")  
  
 *if* to\_evict.id.value *in self*.eviction\_count:  
 *self*.eviction\_count.pop(to\_evict.id.value)  
  
 *if not* bucket.contains(to\_evict.id):  
 *raise* BucketDoesNotContainContactToEvictError(  
 "Bucket does not contain the contact to be evicted."  
 )  
 *else*:  
 bucket.evict\_contact(to\_evict)  
  
 *def* save(*self*, filename: str) -> *None*:  
 *"""  
 Saves DHT to file.  
 """* print(f"[Client] Saving DHT to {filename}...")  
 helpers.make\_sure\_filepath\_exists(filename)  
 *with* open(filename, "wb") *as* output\_file:  
 dill.dump(*self*, file=output\_file)  
 print(f"[Client] Saved DHT to {filename}.")  
  
 @classmethod  
 *def* load(*cls*, filename: str):  
 *"""  
 Loads DHT from file.  
 """* print(f"[Client] Loading DHT from file {filename}...")  
 *with* open(filename, "rb") *as* input\_file:  
 data = dill.load(file=input\_file)  
 print(f"[Client] Loaded DHT from file {filename}.")  
 *return* data  
  
 *def* \_replace\_with\_pending\_contact(*self*, bucket: KBucket) -> *None*:  
 *"""  
 Find a pending contact that goes into the bucket that now has room;  
 that pending contact is no longer pending.  
 :param bucket:  
 :return:  
 """  
 # lock(self.pending\_contacts)* contact: Optional[Contact] = sorted([c *for* c *in self*.pending\_contacts *if  
 self*.node.bucket\_list.get\_kbucket(c.id) == bucket],  
 key=*lambda* c: c.last\_seen)[-1]  
 *if* contact *is not None*:  
 *self*.pending\_contacts.remove(contact)  
 bucket.add\_contact(contact)

### dictionaries.py

*from* typing *import* Callable, TypedDict  
  
*from* kademlia.contact *import* Contact  
*from* kademlia.id *import* ID  
  
  
*class* FindResult(TypedDict):  
 *"""  
 Has elements: contacts, val, found, found\_by  
 """* contacts: list[Contact]  
 val: str | *None* found: bool  
 found\_by: Contact | *None  
  
  
class* ContactQueueItem(TypedDict):  
 key: ID  
 contact: Contact  
 rpc\_call: Callable  
 closer\_contacts: list[Contact]  
 further\_contacts: list[Contact]  
 find\_result: FindResult  
  
  
*class* GetCloserNodesReturn(TypedDict):  
 found: bool  
 found\_by: Contact | *None* val: str | *None  
  
  
class* BaseRequest(TypedDict):  
 protocol: object  
 protocol\_name: str  
 sender: int  
 random\_id: int  
  
  
*class* FindNodeRequest(BaseRequest, TypedDict):  
 key: int  
  
  
*class* FindValueRequest(BaseRequest, TypedDict):  
 key: int  
  
  
*class* PingRequest(BaseRequest, TypedDict):  
 *pass  
  
  
class* StoreRequest(BaseRequest, TypedDict):  
 key: int  
 value: str  
 is\_cached: bool  
 expiration\_time\_sec: int  
  
  
*class* ITCPSubnet(TypedDict):  
 *"""  
 Interface used for TCP Subnetting.  
 """* subnet: int  
  
  
*class* FindNodeSubnetRequest(FindNodeRequest, ITCPSubnet, TypedDict):  
 *pass  
  
  
class* FindValueSubnetRequest(FindValueRequest, ITCPSubnet, TypedDict):  
 *pass  
  
  
class* PingSubnetRequest(PingRequest, ITCPSubnet, TypedDict):  
 *pass  
  
  
class* StoreSubnetRequest(StoreRequest, ITCPSubnet, TypedDict):  
 *pass  
  
  
class* CommonRequest(TypedDict):  
 *"""  
 This includes all possible headers that could be passed.  
 Protocol is type IProtocol, it cannot be type hinted as that because of circular imports.  
 """* protocol: any *# IProtocol* protocol\_name: str  
 random\_id: int  
 sender: int  
 key: int  
 value: str | *None* is\_cached: bool  
 expiration\_time\_sec: int  
  
  
*class* BaseResponse(TypedDict):  
 *"""  
 Has element random\_id (int).  
 """* random\_id: int  
  
  
*class* ErrorResponse(BaseResponse, TypedDict):  
 error\_message: str  
  
  
*class* ContactResponse(TypedDict):  
 contact: int  
 protocol: dict *# Or object?* protocol\_name: dict  
  
  
*class* FindNodeResponse(BaseResponse, TypedDict):  
 contacts: list[ContactResponse]  
  
  
*class* FindValueResponse(TypedDict, BaseResponse):  
 contacts: list[ContactResponse]  
 value: str  
  
  
*class* PingResponse(TypedDict, BaseResponse):  
 *pass  
  
  
class* StoreResponse(BaseResponse):  
 *pass  
  
  
class* StoreValue(TypedDict):  
 *"""  
 Has attributes:  
  
 value: str  
  
 republish\_timestamp: datetime  
  
 expiration\_time: int  
 """* value: str *# | bytes* republish\_timestamp: str  
 expiration\_time: int

### errors.py

*class* DataDecodingError(Exception):  
 *pass  
  
  
class* TooManyContactsError(Exception):  
 *"""Raised when a contact is added to a full k-bucket."""  
 pass  
  
  
class* OutOfRangeError(Exception):  
 *"""Raised when a contact is added to a k-bucket that is out of range."""  
 pass  
  
  
class* OurNodeCannotBeAContactError(Exception):  
 *"""Raised when a contact added has the same ID as the client."""  
  
  
class* AllKBucketsAreEmptyError(Exception):  
 *"""Raised when no KBuckets can be iterated through."""  
  
  
class* SendingQueryToSelfError(Exception):  
 *"""Raised when a Query (RPC Call) is sent to ourselves."""  
 pass  
  
  
class* SenderIsSelfError(Exception):  
 *"""Raised when trying to send certain RPC commands, if sender is us."""  
 pass  
  
  
class* BucketDoesNotContainContactToEvictError(Exception):  
 *pass  
  
  
class* RPCError(Exception):  
 *"""  
 Possible errors for RPC commands.  
 """  
  
 def* \_\_init\_\_(*self*,  
 error\_message: str | *None* = *None*,  
 timeout\_error: bool = *False*,  
 id\_mismatch\_error: bool = *False*,  
 peer\_error: bool = *False*,  
 peer\_error\_message: str | *None* = *None* ):  
 *"""  
 Initialises an RPCError method – having all these error types together allows checking for RPCErrors  
 very easy, and still readable.  
 :param error\_message:  
 :param timeout\_error:  
 :param id\_mismatch\_error:  
 :param peer\_error:  
 :param peer\_error\_message:  
 """* super().\_\_init\_\_(error\_message)  
 *self*.protocol\_error\_message: str | *None* = error\_message  
  
 *if* error\_message:  
 *self*.protocol\_error = *True  
 else*:  
 *self*.protocol\_error = *False  
  
 self*.timeout\_error = timeout\_error  
 *self*.id\_mismatch\_error = id\_mismatch\_error  
 *self*.peer\_error = peer\_error  
 *self*.peer\_error\_message: str | *None* = peer\_error\_message  
  
 *if self*.peer\_error\_message *and not self*.peer\_error:  
 *raise* ValueError("Parameter peer error message requires a peer error.")  
  
 *def* has\_error(*self*) -> bool:  
 *"""  
 Returns True if any type of error is true, else False.  
 :return:  
 """* print(*self*.timeout\_error, *self*.protocol\_error, *self*.id\_mismatch\_error, *self*.peer\_error)  
 *return self*.timeout\_error *or* \  
 *self*.protocol\_error *or* \  
 *self*.id\_mismatch\_error *or* \  
 *self*.peer\_error  
  
 *def* \_\_str\_\_(*self*):  
 *"""  
 Returns error message, or “No error” if there is none.  
 :return:  
 """  
 if self*.has\_error():  
 *if self*.protocol\_error:  
 *return* f"Protocol error: {*self*.protocol\_error\_message}"  
 *elif self*.peer\_error:  
 *return* f"Peer error: {*self*.peer\_error\_message}"  
 *elif self*.timeout\_error:  
 *return* "Timeout error."  
 *elif self*.id\_mismatch\_error:  
 *return* "ID mismatch error."  
 *else*:  
 *return* "Unknown error."  
 *else*:  
 *return* "No error."  
  
 @classmethod  
 *def* no\_error(*cls*):  
 *return cls*()  
  
  
*class* ValueCannotBeNoneError(Exception):  
 *"""  
 Raised when a value is None, when everything was meant to have gone OK.  
 There is a risk of this being purposely triggered maliciously to shut down nodes on the network.  
 """  
  
  
class* UnknownRequestError(Exception):  
 *pass  
  
  
class* IncorrectProtocolError(Exception):  
 *pass*

### helpers.py

*import* os  
*from* hashlib *import* sha1  
*import* random  
*import* socket  
  
*from* kademlia.contact *import* Contact  
*from* kademlia.node *import* Node  
*from* kademlia.id *import* ID  
*from* kademlia.storage *import* VirtualStorage  
  
  
*def* empty\_node():  
 *"""  
 For testing.  
 :return:  
 """  
 return* Node(Contact(id=ID(0)), storage=VirtualStorage())  
  
  
*def* random\_node():  
 *return* Node(Contact(id=ID.random\_id()), storage=VirtualStorage())  
  
  
*def* select\_random(arr: list, freq: int) -> list:  
 *return* random.sample(arr, freq)  
  
  
*def* get\_closest\_number\_index(numbers, target):  
 closest\_index = 0  
 closest\_difference = abs(numbers[0] - target)  
  
 *for* i *in* range(1, len(numbers)):  
 difference = abs(numbers[i] - target)  
 *if* difference < closest\_difference:  
 closest\_difference = difference  
 closest\_index = i  
  
 *return* closest\_index  
  
  
*def* convert\_file\_to\_key(filename: str) -> ID:  
 sha1\_hash = sha1()  
 *with* open(filename, 'rb') *as* file:  
 *while True*:  
 data = file.read(4096) *# Read data from the file in chunks  
 if not* data:  
 *break* sha1\_hash.update(data) *# Update the hash object with the read data* digest = int(sha1\_hash.hexdigest(), 16)  
 *return* ID(digest)  
  
  
*def* make\_sure\_filepath\_exists(filename: str) -> *None*:  
 *if* os.path.isabs(filename):  
 print(f"[DEBUG] Path {filename} is absolute.")  
 path = filename  
 *else*:  
 print(f"[DEBUG] Path {filename} is not absolute.")  
 path = os.path.join(os.getcwd(), filename)  
 print(f"[DEBUG] Absolute version is {path}")  
 *if not* os.path.exists(path):  
 print(f"[DEBUG] Path does not exist.")  
 dirname = os.path.dirname(path)  
 *if* dirname:  
 *if not* os.path.exists(dirname):  
 os.mkdir(dirname)  
 *else*:  
 print("[DEBUG] Path already existed.")  
  
  
*def* port\_is\_free(port: int) -> bool:  
 *"""  
 Returns if a port is free on localhost.  
 :param port: Port to be checked  
 :return: if it's free.  
 """  
  
 try*:  
 *with* socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) *as* sock:  
 sock.bind(('localhost', port))  
 *return True  
 except* (OSError, PermissionError):  
 *return False  
  
  
def* get\_valid\_port(default\_tried=*False*,  
 lower\_bound=1024, upper\_bound=65535) -> int:  
 *"""  
 Gets a valid port on localhost.  
 """  
 if* lower\_bound > upper\_bound:  
 *raise* ValueError("Port lower bound cannot be greater than port upper bound.")  
  
 *if not* default\_tried:  
 port = 7124 *# Default port I wish to use.* default\_tried = *True  
 else*:  
 port = random.randint(lower\_bound, upper\_bound)  
 *if* port\_is\_free(port):  
 *return* port  
 *else*:  
 *return* get\_valid\_port(default\_tried=default\_tried)

### id.py

*import* random  
*from* math *import* ceil, log  
  
*from* kademlia.constants *import* Constants  
  
  
*class* ID:  
  
 *def* \_\_init\_\_(*self*, value: int):  
 *"""  
 Kademlia node ID: This is an integer from 0 to 2^160 - 1  
  
 Args:  
 value: (int) ID decimal value  
 """  
  
 self*.MAX\_ID = 2 \*\* Constants.ID\_LENGTH\_BITS  
 *self*.MIN\_ID = 0  
 *if not* (*self*.MAX\_ID > value >= *self*.MIN\_ID): *# ID can be 0, this is used in unit tests.  
 raise* ValueError(  
 f"ID {value} is out of range - must a positive integer less than 2^160."  
 )  
 *self*.value = value  
  
 *def* hex(*self*) -> str:  
 *return* hex(*self*.value)  
  
 *def* decimal(*self*) -> int:  
 *return self*.value  
  
 *def* bin(*self*) -> str:  
 *"""  
 Returns big-endian value in binary - this does not include a 0b tag at the start.  
 :return: Returns the binary value as a string, with length Constants.B by default  
 """* binary = bin(*self*.value)[2:]  
 number\_of\_zeroes\_to\_add = ceil(log(*self*.MAX\_ID, 2)) - len(binary)  
 padded\_binary = number\_of\_zeroes\_to\_add \* "0" + binary  
 *return* padded\_binary  
  
 *# def set\_bit(self, bit: int) -> None:  
 # """  
 # Sets a given bit to 1, Little endian. (set\_bit(0) sets smallest bit to 0)  
 # :param bit: bit to be set.  
 # :return: Nothing  
 # """  
 # self.little\_endian\_bytes()[bit] = "1"  
  
 def* big\_endian\_bytes(*self*) -> list[str]:  
 *"""  
 Returns the padded ID in big-endian binary - largest bit is at index 0.  
 """  
 return* [x *for* x *in self*.bin()]  
  
 *def* little\_endian\_bytes(*self*) -> list[str]:  
 *"""  
 Returns the padded ID in little-endian binary - smallest bit is at index 0.  
 """  
 return self*.big\_endian\_bytes()[::-1]  
  
 *def* \_\_xor\_\_(*self*, val) -> int:  
 *if* isinstance(val, ID):  
 *return self*.value ^ val.value  
 *else*:  
 *return self*.value ^ val  
  
 *def* \_\_eq\_\_(*self*, val) -> bool:  
 *if* isinstance(val, ID):  
 *return self*.value == val.value  
 *else*:  
 *return self*.value == val  
  
 *def* \_\_ge\_\_(*self*, val) -> bool:  
 *if* isinstance(val, ID):  
 *return self*.value >= val.value  
 *else*:  
 *return self*.value >= val  
  
 *def* \_\_le\_\_(*self*, val) -> bool:  
 *if* isinstance(val, ID):  
 *return self*.value <= val.value  
 *else*:  
 *return self*.value <= val  
  
 *def* \_\_lt\_\_(*self*, val) -> bool:  
 *if* isinstance(val, ID):  
 *return self*.value < val.value  
 *else*:  
 *return self*.value < val  
  
 *def* \_\_gt\_\_(*self*, val) -> bool:  
 *if* isinstance(val, ID):  
 *return self*.value > val.value  
 *else*:  
 *return self*.value > val  
  
 *def* \_\_str\_\_(*self*) -> str:  
 *return* str(*self*.value)  
  
 *def* \_\_repr\_\_(*self*) -> str:  
 *return* str(*self*.value)  
  
 @classmethod  
 *def* max(*cls*):  
 *"""  
 Returns max ID.  
 :return: max ID.  
 """  
 return* ID(2\*\*160 - 1)  
  
 @classmethod  
 *def* mid(*cls*):  
 *"""  
 returns middle of the road ID  
 :return: middle ID.  
 """  
 return* ID(2\*\*159 - 1) *# Should this be ID(2\*\*159)? But then ID(1) ^ ID.mid() > ID.mid() ^ ID.max()* @classmethod  
 *def* min(*cls*):  
 *"""  
 Returns minimum ID.  
 :return: minimum ID.  
 """  
 return* ID(0)  
  
 @classmethod  
 *def* random\_id\_within\_bucket\_range(*cls*, bucket):  
 *"""  
 Returns an ID within the range of the bucket's low and high range.  
 THIS IS NOT AN ID IN THE BUCKETS CONTACT LIST!  
 (I mean it could be but shush)  
  
 :param bucket: bucket to be searched  
 :return: random ID in bucket.  
 """  
 return* ID(bucket.low() + random.randint(0, bucket.high() - bucket.low()))  
  
 @classmethod  
 *def* random\_id(*cls*, low=0, high=2\*\*160, seed=*None*):  
 *"""  
 Generates a random ID, including both endpoints.  
  
 FOR TESTING PURPOSES.  
 Generating random ID's this way will not perfectly spread the prefixes,  
 this is a maths law I've forgotten - due to the small scale of this  
 I don't particularly see the need to perfectly randomise this.  
  
 If I do though, here's how it would be done:  
 - Randomly generate each individual bit, then concatenate.  
 """  
 if* seed:  
 random.seed(seed)  
 *return* ID(random.randint(low, high))

### interfaces.py

*from* abc *import* abstractmethod  
*from* datetime *import* datetime  
  
*from* kademlia.errors *import* RPCError  
*from* kademlia.id *import* ID  
  
  
*class* IStorage:  
 *"""Interface which 'abstracts the storage mechanism for key-value pairs.''"""* @abstractmethod  
 *def* contains(*self*, key: ID) -> bool:  
 *"""  
 Returns if the given key is contained in the storage object.  
 :param key:  
 :return:  
 """  
 pass* @abstractmethod  
 *def* try\_get\_value(*self*, key: ID) -> tuple[bool, str]:  
 *"""  
 Attempts to get a value from a key-value pair, returns ‘False, “”’ if it is not there; returns  
 ‘True, value’ if it is there.  
 :param key:  
 :return:  
 """  
 pass* @abstractmethod  
 *def* get(*self*, key: ID | int) -> str:  
 *"""  
 Tries to return value from key-value pair, given key.  
 :param key:  
 :return: StoreValue  
 """  
 pass* @abstractmethod  
 *def* get\_timestamp(*self*, key: int) -> datetime:  
 *"""  
 Returns timestamp of key-value pair, given key.  
 :param key:  
 :return:  
 """  
 pass* @abstractmethod  
 *def* set(*self*, key: ID, value: str, expiration\_time\_sec: int = 0) -> *None*:  
 *"""  
 Sets key-value pair with expiration time.  
 :param key:  
 :param value:  
 :param expiration\_time\_sec:  
 :return:  
 """  
 pass* @abstractmethod  
 *def* get\_expiration\_time\_sec(*self*, key: int) -> int:  
 *"""  
 Gets expiration time from key-value pair, given key.  
 :param key:  
 :return:  
 """  
 pass* @abstractmethod  
 *def* remove(*self*, key: int) -> *None*:  
 *"""  
 Removes key-value pair from the storage object, given key.  
 :param key:  
 :return:  
 """  
 pass* @abstractmethod  
 *def* get\_keys(*self*) -> list[int]:  
 *"""  
 Returns all keys from key-value pairs stored in the storage object.  
 :return:  
 """  
 pass* @abstractmethod  
 *def* touch(*self*, key: int) -> *None*:  
 *"""  
 Sets timestamp of key-value to current time, given key.  
 :param key:  
 :return:  
 """  
 pass  
  
  
class* IProtocol:  
 *"""  
 Interface for all protocols to follow.  
 """  
 def* \_\_init\_\_(*self*):  
 *self*.node = *None* @abstractmethod  
 *def* ping(*self*, sender) -> RPCError:  
 *"""  
 Handles an incoming ping request from “sender”, returns an RPCError object.  
 :param sender:  
 :return:  
 """  
 pass* @abstractmethod  
 *def* find\_node(*self*, sender, key: ID) -> tuple[list, RPCError]:  
 *"""  
 Attempts to find K close nodes to key, returning them and an RPCError object.  
 :param sender:  
 :param key:  
 :return:  
 """  
 pass* @abstractmethod  
 *def* find\_value(*self*, sender, key: ID) -> tuple[list, str, RPCError]:  
 *"""  
 Attempts to find value from key-value pair, if it cannot be found, a list of K  
 closer contacts are returned. An RPCError object is returned as well to indicate  
 any errors that have occurred.  
  
 :param sender:  
 :param key:  
 :return:  
 """  
 pass* @abstractmethod  
 *def* store(*self*, sender, key: ID, val: str, is\_cached: bool = *False*, exp\_time\_sec: int = 0) -> RPCError:  
 *"""  
 Attempts to save a key-value pair to storage, it caches it instead of  
 storing if is\_stored, and it expires after exp\_time\_sec.  
  
 :param sender:  
 :param key:  
 :param val:  
 :param is\_cached:  
 :param exp\_time\_sec:  
 :return:  
 """  
 pass*

### locker.py

*from* threading *import* Lock  
  
  
*class* WithLock:  
 *"""  
 Lock object that can be used in "with" statements.  
 Example usage:  
 lock = threading.Lock()  
 with WithLock(lock):  
 do\_stuff()  
 do\_more\_stuff()  
 Based from the following code:  
 https://www.bogotobogo.com/python/Multithread/python\_multithreading\_Synchronization\_Lock\_Objects\_Acquire\_Release.php  
 https://www.geeksforgeeks.org/with-statement-in-python/  
 """  
  
 def* \_\_init\_\_(*self*, lock: Lock) -> *None*:  
 *"""  
 Creates lock object to be used in \_\_enter\_\_ and \_\_exit\_\_.  
 """  
 self*.\_\_lock = lock  
  
 *def* \_\_enter\_\_(*self*) -> *None*:  
 *"""  
 Change the state to locked and returns immediately.  
 """  
 self*.\_\_lock.acquire()  
  
 *def* \_\_exit\_\_(*self*, exc\_type, exc\_value, traceback) -> *None*:  
 *"""  
 Changes the state to unlocked; this is called from another thread.  
 """  
 self*.\_\_lock.release()

### my\_queues.py

*from* typing *import* Optional  
  
  
*class* InfiniteLinearQueue:  
 *def* \_\_init\_\_(*self*):  
 *"""  
 Makes a linear queue with no maximum size.  
  
 You may ask, what's the point of this? isn't this just a normal list?  
 Basically, it is; but you have a dequeue method.  
 """  
 self*.\_\_items = []  
  
 *def* is\_empty(*self*):  
 *"""  
 Returns if the queue is empty or not.  
 :return:  
 """  
 return* len(*self*.\_\_items) == 0  
  
 *def* enqueue(*self*, item) -> *None*:  
 *"""  
 Adds an item to the queue.  
 :param item:  
 :return:  
 """  
 self*.\_\_items.append(item)  
  
 *def* dequeue(*self*) -> Optional[any]:  
 *"""  
 Removes an item to the queue and returns it – returns None if it is not in the queue.  
 :return:  
 """  
 if not self*.is\_empty():  
 *return self*.\_\_items.pop(0)  
 *else*:  
 *return None  
  
  
class* LinearQueue(object):  
  
 *def* \_\_init\_\_(*self*, queue\_size: int):  
 *"""  
 Initialises the queue with a given maximum size.  
 :param queue\_size:  
 """  
 self*.\_\_queue = [*None*] \* queue\_size  
 *self*.\_\_front = 0  
 *self*.\_\_rear = -1 *# enQueue is never called for -1, as it increases by 1 before calling it  
 self*.\_\_size = 0  
 *self*.\_\_max\_size = queue\_size  
  
 *def* enqueue(*self*, item):  
 *"""  
 Adds item to the tail of the queue. Returns True if success, False if failure.  
 :param item:  
 :return:  
 """  
 if self*.is\_full():  
 *return False  
 else*:  
 *self*.\_\_rear += 1  
 *self*.\_\_queue[*self*.\_\_rear] = item  
 *self*.\_\_size += 1  
 *return True  
  
 def* dequeue(*self*):  
 *"""  
 Removes an item from the front of the list and returns it. Returns None if the list is empty.  
 :return:  
 """  
 if self*.is\_empty():  
 *return None  
 else*:  
 *self*.\_\_size -= 1  
 item = *self*.\_\_queue[*self*.\_\_front]  
 *self*.\_\_front += 1  
 *return* item  
  
 *def* is\_full(*self*):  
 *"""  
 Returns if the list is full.  
 :return:  
 """  
 return self*.\_\_rear >= *self*.\_\_max\_size - 1 *# Returns boolean  
  
 def* is\_empty(*self*):  
 *"""  
 Returns if the list is empty.  
 :return:  
 """  
 return self*.\_\_size == 0 *# Returns boolean  
  
 def* \_\_str\_\_(*self*):  
 *"""  
 Prints each element of the queue separated by whitespace.  
 :return:  
 """* output = ""  
 *for* index *in* range(*self*.\_\_front, *self*.\_\_rear + 1):  
 output += str(*self*.\_\_queue[index]) + " "  
 *return* output[:-1]

### networking.py

*import* threading  
*from* time *import* sleep  
*from* typing *import* Optional, TypedDict, Callable  
*from* http.server *import* BaseHTTPRequestHandler, ThreadingHTTPServer  
  
*import* kademlia.pickler *as* pickler  
*from* kademlia.constants *import* Constants  
*from* kademlia.dictionaries *import* PingRequest, StoreRequest, FindNodeRequest, FindValueRequest, ErrorResponse, \  
 CommonRequest, PingSubnetRequest, StoreSubnetRequest, FindNodeSubnetRequest, FindValueSubnetRequest  
*from* kademlia.errors *import* IncorrectProtocolError  
*from* kademlia.id *import* ID  
*from* kademlia.node *import* Node  
*from* kademlia.protocols *import* TCPProtocol  
  
  
*class* BaseServer(ThreadingHTTPServer):  
 *def* \_\_init\_\_(*self*, server\_address: tuple[str, int], request\_handler\_class):  
 print(f"server address: {server\_address}")  
 ThreadingHTTPServer.\_\_init\_\_(  
 *self*,  
 server\_address=server\_address,  
 RequestHandlerClass=request\_handler\_class  
 )  
  
 *self*.routing\_methods: dict[str, type] = {  
 "/ping": PingRequest, *# "ping" should refer to type PingRequest* "/store": StoreRequest, *# "store" should refer to type StoreRequest* "/find\_node": FindNodeRequest, *# "find\_node" should refer to type FindNodeRequest* "/find\_value": FindValueRequest *# "find\_value" should refer to type FindValueRequest* }  
  
 *def* start(*self*) -> *None*:  
 *"""  
 Starts the server.  
 :return:  
 """* print("[Server] Starting server...")  
 *self*.serve\_forever()  
  
 *def* stop(*self*):  
 *"""  
 Stops the server.  
 :return:  
 """* print("[Server] Stopping server...")  
 *self*.shutdown()  
 *self*.server\_close()  
  
 *def* thread\_start(*self*) -> threading.Thread:  
 *"""  
 Starts the server on a specific thread that is returned –  
 this is probably obsolete now that ThreadingHTTPServer is used, instead of HTTPServer.  
 :return: Thread the server is running on  
 """* thread = threading.Thread(target=*self*.start)  
 thread.start()  
 *return* thread  
  
 *def* thread\_stop(*self*, thread: threading.Thread) -> *None*:  
 *"""  
 Stops the server on a given thread.  
 If the thread is invalid, the server will still shut  
 :param thread:  
 :return:  
 """  
 self*.shutdown()  
 *self*.server\_close()  
 thread.join() *# wait for the thread to finish.* print("[Server] Server stopped.")  
  
  
*class* HTTPSubnetRequestHandler(BaseHTTPRequestHandler):  
  
 *def* \_common\_request\_handler(*self*,  
 method\_name: str, common\_request: CommonRequest, node):  
 old\_self\_instance = *self # To prevent other threads overwriting it,  
 # lock isn't used because I don't want to make the program wait.  
  
 # Test what happens if a node does not respond  
 if* Constants.DEBUG:  
 *if* node.our\_contact.protocol.type == "TCPSubnetProtocol":  
 *if not* node.our\_contact.protocol.responds:  
 *# Exceeds 500ms timeout* print("[Server] Does not respond, sleeping for timeout.")  
 sleep(1)  
  
 *try*:  
 method: Callable = getattr(node, method\_name)  
 *# Calls method, eg: server\_store.* response = method(common\_request)  
 encoded\_response = pickler.encode\_data(response)  
 print("[Server] Sending encoded 200: ", response)  
 old\_self\_instance.send\_response(code=200)  
  
 *# print("Adding headers... - Is wfile closed:", self.wfile.closed)* old\_self\_instance.send\_header("Content-Type", "application/octet-stream")  
 old\_self\_instance.end\_headers()  
 *# print("Finished headers - Is wfile closed:", self.wfile.closed)  
  
 # print("Writing 200...", self.wfile.closed)  
 try*:  
 old\_self\_instance.wfile.write(encoded\_response)  
 print("[Server] Writing response success!")  
 *except* ConnectionRefusedError:  
 print("[ERROR] [Server] Connection refused by client - we may have timed out.")  
 *except* Exception *as* e:  
 print("[Server] Exception sending response:", e)  
  
 *except* Exception *as* e:  
 print("[Server] Exception sending response.")  
 error\_response: ErrorResponse = ErrorResponse(  
 error\_message=str(e),  
 random\_id=ID.random\_id()  
 )  
 print("[Server] Sending encoded 400:", error\_response)  
 encoded\_response = pickler.encode\_data(error\_response)  
  
 old\_self\_instance.send\_header("Content-Type", "application/octet-stream")  
 old\_self\_instance.end\_headers()  
 old\_self\_instance.send\_response(code=400) *# , message=encoded\_response.decode("latin1"))  
  
 try*:  
 old\_self\_instance.wfile.write(encoded\_response)  
 *except* ConnectionRefusedError:  
 print("[ERROR] [Server] Connection refused by client - we may have timed out.")  
 *except* Exception *as* e:  
 print("[Server] Exception sending response:", e)  
  
 *# old\_self\_instance.wfile.close()  
 # finally:  
 # if not old\_self\_instance.wfile.closed:  
 # old\_self\_instance.wfile.close()  
 # else:  
 # print("[Server] Response body was already closed! (What on earth, something's gone wrong!)")  
  
 def* do\_POST(*self*):  
 print("[Server] POST Received.")  
 routing\_methods = {  
 "/ping": PingRequest, *# "ping" should refer to type PingRequest* "/store": StoreRequest, *# "store" should refer to type StoreRequest* "/find\_node": FindNodeRequest, *# "find\_node" should refer to type FindNodeRequest* "/find\_value": FindValueRequest *# "find\_value" should refer to type FindValueRequest* }  
  
 content\_length = int(*self*.headers['Content-Length'])  
 encoded\_request: bytes = *self*.rfile.read(content\_length)  
 *# encoded\_request: bytes = self.rfile.read()* decoded\_request: dict = pickler.decode\_data(encoded\_request)  
 *# print("[Server] Request received:", decoded\_request)* request\_dict = decoded\_request  
 path: str = *self*.path  
 *# Remove "/"  
 # Prefix our call with "server\_" so that the method name is unambiguous.* method\_name: str = "server\_" + path[1:] *# path.substring(2)  
 # What type is the request?  
 try*:  
 *# path is something like /ping or /find\_node* request\_type: Optional[TypedDict] = routing\_methods[path]  
 *except* KeyError:  
 request\_type: Optional[TypedDict] = *None  
  
 # if we know what the request wants (if it's a ping/find\_node RPC etc.)  
 if* request\_type:  
 subnet: int = request\_dict["subnet"]  
 common\_request: CommonRequest = CommonRequest(  
 protocol=request\_dict.get("protocol"),  
 protocol\_name=request\_dict.get("protocol\_name"),  
 random\_id=request\_dict.get("random\_id"),  
 sender=request\_dict.get("sender"),  
 key=request\_dict.get("key"),  
 value=request\_dict.get("value"),  
 is\_cached=request\_dict.get("is\_cached"),  
 expiration\_time\_sec=request\_dict.get("expiration\_time\_sec")  
 )  
  
 *# If we know the node on the subnet, this should always happen right?  
 # Because this is for testing on the same PC.* node = *self*.server.subnets.get(subnet) *# should be valid if inheriting from SubnetServer?  
 if* node:  
 print("[Server] Request called:", node.bucket\_list.buckets)  
 *self*.\_common\_request\_handler(method\_name, common\_request, node)  
 *# print("Starting thread...")  
 # new\_thread = threading.Thread(  
 # target=self.common\_request\_handler,  
 # args=(method\_name, common\_request, node)  
 # )  
 # new\_thread.start()  
  
 else*:  
 print("[Server] Subnet node not found.")  
 encoded\_response = pickler.encode\_data({"error\_message": "Subnet node not found."})  
 *self*.send\_header("Content-Type", "application/octet-stream")  
 *self*.end\_headers()  
 *self*.send\_response(400)  
 *try*:  
 *self*.wfile.write(encoded\_response)  
 *except* ConnectionRefusedError:  
 print("[ERROR] [Server] Connection refused by client - we may have timed out.")  
 *except* Exception *as* e:  
 print("[Server] Exception sending response:", e)  
  
 *# context.close\_connection = True  
  
  
class* TCPSubnetServer(BaseServer):  
 *def* \_\_init\_\_(*self*, server\_address: tuple[str, int]):  
 super().\_\_init\_\_(  
 server\_address=server\_address,  
 request\_handler\_class=HTTPSubnetRequestHandler  
 )  
 *self*.subnets: dict = {}  
 *self*.routing\_methods: dict[str, type] = {  
 "/ping": PingSubnetRequest, *# "ping" should refer to type PingSubnetRequest* "/store": StoreSubnetRequest, *# "store" should refer to type StoreSubnetRequest* "/find\_node": FindNodeSubnetRequest, *# "find\_node" should refer to type FindNodeSubnetRequest* "/find\_value": FindValueSubnetRequest *# "find\_value" should refer to type FindValueSubnetRequest* }  
  
 *def* register\_protocol(*self*, subnet: int, node):  
 *self*.subnets[subnet] = node  
  
  
*class* HTTPRequestHandler(BaseHTTPRequestHandler):  
 *def* \_common\_request\_handler(*self*,  
 method\_name: str, common\_request: CommonRequest, node):  
 old\_self\_instance = *self # To prevent other threads overwriting it,  
 # lock isn't used because I don't want to make the program wait.  
  
 # Test what happens if a node does not respond  
 if* Constants.DEBUG:  
 *if* node.our\_contact.protocol.type == "TCPSubnetProtocol":  
 *if not* node.our\_contact.protocol.responds:  
 *# Exceeds 500ms timeout* print("[Server] Does not respond, sleeping for timeout.")  
 sleep(1)  
  
 *try*:  
 method: Callable = getattr(node, method\_name)  
 *# Calls method, eg: server\_store.* response = method(common\_request)  
 encoded\_response = pickler.encode\_data(response)  
 print("[Server] Sending encoded 200: ", response)  
 old\_self\_instance.send\_response(code=200)  
  
 *# print("Adding headers... - Is wfile closed:", self.wfile.closed)* old\_self\_instance.send\_header("Content-Type", "application/octet-stream")  
 old\_self\_instance.end\_headers()  
 *# print("Finished headers - Is wfile closed:", self.wfile.closed)  
  
 # print("Writing 200...", self.wfile.closed)]  
 try*:  
 old\_self\_instance.wfile.write(encoded\_response)  
 print("[Server] Writing response success!")  
 *except* ConnectionRefusedError:  
 print("[ERROR] [Server] Connection refused by client - we may have timed out.")  
 *except* Exception *as* e:  
 print("[Server] Exception sending response:", e)  
  
  
 *except* Exception *as* e:  
 print("[Server] Exception sending response.")  
 error\_response: ErrorResponse = ErrorResponse(  
 error\_message=str(e),  
 random\_id=ID.random\_id()  
 )  
 print("[Server] Sending encoded 400:", error\_response)  
 encoded\_response = pickler.encode\_data(error\_response)  
  
 old\_self\_instance.send\_header("Content-Type", "application/octet-stream")  
 old\_self\_instance.end\_headers()  
 old\_self\_instance.send\_response(code=400) *# , message=encoded\_response.decode("latin1"))  
 try*:  
 old\_self\_instance.wfile.write(encoded\_response)  
 *except* ConnectionRefusedError:  
 print("[ERROR] [Server] Connection refused by client - we may have timed out.")  
 *except* Exception *as* e:  
 print("[Server] Exception sending response:", e)  
  
 *# old\_self\_instance.wfile.close()  
 # finally:  
 # if not old\_self\_instance.wfile.closed:  
 # old\_self\_instance.wfile.close()  
 # else:  
 # print("[Server] Response body was already closed! (What on earth, something's gone wrong!)")  
  
 def* do\_POST(*self*):  
 print("[Server] POST Received.")  
  
 routing\_methods = {  
 "/ping": PingRequest, *# "ping" should refer to type PingRequest* "/store": StoreRequest, *# "store" should refer to type StoreRequest* "/find\_node": FindNodeRequest, *# "find\_node" should refer to type FindNodeRequest* "/find\_value": FindValueRequest *# "find\_value" should refer to type FindValueRequest* }  
  
 content\_length = int(*self*.headers['Content-Length'])  
 encoded\_request: bytes = *self*.rfile.read(content\_length)  
 *# encoded\_request: bytes = self.rfile.read()* decoded\_request: dict = pickler.decode\_data(encoded\_request)  
 *# print("[Server] Request received:", decoded\_request)* request\_dict = decoded\_request  
 path: str = *self*.path  
 *# Remove "/"  
 # Prefix our call with "server\_" so that the method name is unambiguous.* method\_name: str = "server\_" + path[1:] *# path.substring(2)  
 # What type is the request?  
 try*:  
 *# path is something like /ping or /find\_node* request\_type: Optional[TypedDict] = routing\_methods[path]  
 *except* KeyError:  
 request\_type: Optional[TypedDict] = *None  
  
 # if we know what the request wants (if it's a ping/find\_node RPC etc.)  
 if* request\_type:  
 common\_request: CommonRequest = CommonRequest(  
 protocol=request\_dict.get("protocol"),  
 protocol\_name=request\_dict.get("protocol\_name"),  
 random\_id=request\_dict.get("random\_id"),  
 sender=request\_dict.get("sender"),  
 key=request\_dict.get("key"),  
 value=request\_dict.get("value"),  
 is\_cached=request\_dict.get("is\_cached"),  
 expiration\_time\_sec=request\_dict.get("expiration\_time\_sec")  
 )  
  
 node = *self*.server.node  
 *if* node:  
 print("[Server] Request called:", node.bucket\_list.buckets)  
 *self*.\_common\_request\_handler(method\_name, common\_request, node)  
 *# print("Starting thread...")  
 # new\_thread = threading.Thread(  
 # target=self.common\_request\_handler,  
 # args=(method\_name, common\_request, node)  
 # )  
 # new\_thread.start()  
  
 else*:  
 print("[Server] Node not found.")  
 encoded\_response = pickler.encode\_data({"error\_message": "Node not found."})  
 *self*.send\_header("Content-Type", "application/octet-stream")  
 *self*.end\_headers()  
 *self*.send\_response(400)  
 *try*:  
 *self*.wfile.write(encoded\_response)  
 *except* ConnectionRefusedError:  
 print("[ERROR] [Server] Connection refused by client - we may have timed out.")  
 *except* Exception *as* e:  
 print("[Server] Exception sending response:", e)  
  
 *# context.close\_connection = True  
  
  
class* TCPServer(BaseServer):  
 *def* \_\_init\_\_(*self*, node: Node):  
 *"""  
 Creates a server using TCP, based on a Threading HTTP Server from http.server, the  
 given node provides the IP and port tuple to start the server.  
 :param node:  
 """  
 self*.node = node  
 *if* isinstance(*self*.node.our\_contact.protocol, TCPProtocol):  
 server\_address: tuple[str, int] = ("127.0.0.1", *self*.node.our\_contact.protocol.port)  
 *else*:  
 *raise* IncorrectProtocolError("Invalid protocol.")  
  
 super().\_\_init\_\_(  
 server\_address=server\_address,  
 request\_handler\_class=HTTPRequestHandler  
 )

### node.py

*from* kademlia.buckets *import* BucketList  
*from* kademlia.constants *import* Constants  
*from* kademlia.contact *import* Contact  
*from* kademlia.dictionaries *import* CommonRequest  
*from* kademlia.errors *import* RPCError, SenderIsSelfError, SendingQueryToSelfError  
*from* kademlia.id *import* ID  
*from* kademlia.interfaces *import* IProtocol, IStorage  
*from* kademlia.storage *import* VirtualStorage  
  
  
*class* Node:  
  
 *def* \_\_init\_\_(*self*,  
 contact: Contact,  
 storage: IStorage,  
 cache\_storage=*None*):  
 *"""  
 Initialises the node object. This object is used to represent a device on the  
 network – contains IP, port, ID, storage, cache\_storage, and the master DHT object controlling it.  
 :param contact:  
 :param storage:  
 :param cache\_storage:  
 """  
  
 if not* cache\_storage *and not* Constants.DEBUG:  
 *raise* ValueError(  
 "cache\_storage must be supplied to type node if debug mode is not enabled."  
 )  
  
 *self*.our\_contact: Contact = contact  
 *self*.storage: IStorage = storage  
  
 *# VirtualStorage will only be created by  
 self*.cache\_storage: IStorage = cache\_storage *if* cache\_storage *else* VirtualStorage()  
 *self*.dht = *None # This should never be None  
 self*.bucket\_list = BucketList(contact)  
  
 *def* ping(*self*, sender: Contact) -> Contact:  
 *"""  
 Someone is pinging us.  
 Register the contact and respond with our contact.  
 """  
 if* sender.id.value == *self*.our\_contact.id.value:  
 *raise* SendingQueryToSelfError(  
 "Sender of ping RPC cannot be ourself."  
 )  
 *self*.send\_key\_values\_if\_new\_contact(sender)  
 *self*.bucket\_list.add\_contact(sender)  
  
 *return self*.our\_contact  
  
 *def* store(*self*,  
 key: ID,  
 sender: Contact,  
 val: str,  
 is\_cached: bool = *False*,  
 expiration\_time\_sec: int = 0) -> *None*:  
 *"""  
 Stores a key-value pair in main/cache storage. This adds the sender to our bucket list – so it  
 will error if the sender is ourselves. Then it will send key values if the contact is new  
 (and not is\_cached), then it will save the key-value pair in the corresponding storage object.  
  
 :param key:  
 :param sender:  
 :param val:  
 :param is\_cached:  
 :param expiration\_time\_sec:  
 :return:  
 """  
  
 if* sender.id.value == *self*.our\_contact.id.value:  
 *raise* SenderIsSelfError("Sender should not be ourself.")  
  
 *# add sender to bucket\_list (updating bucket list like how it is in spec.)  
 self*.bucket\_list.add\_contact(sender)  
  
 *if* is\_cached:  
 *self*.cache\_storage.set(key, val, expiration\_time\_sec)  
 *else*:  
 *self*.send\_key\_values\_if\_new\_contact(sender)  
 *self*.storage.set(key, val, Constants.EXPIRATION\_TIME\_SEC)  
  
 *def* find\_node(*self*, key: ID,  
 sender: Contact) -> tuple[list[Contact], str | *None*]:  
 *"""  
 Finds K close contacts to a given ID, whilst excluding the sender.  
 It also adds the sender if it hasn't seen it before.  
 :param key: K close contacts are found near this ID.  
 :param sender: Contact to be excluded and added if new.  
 :return: list of K (or less) contacts near the key  
 """  
  
 # managing sender  
 if* sender.id == *self*.our\_contact.id:  
 *raise* SendingQueryToSelfError("Sender cannot be ourselves.")  
 *self*.send\_key\_values\_if\_new\_contact(sender)  
 *self*.bucket\_list.add\_contact(sender)  
  
 *# actually finding nodes  
 # print([len(b.contacts) for b in self.bucket\_list.buckets])* contacts = *self*.bucket\_list.get\_close\_contacts(key=key,  
 exclude=sender.id)  
 *# print(f"contacts: {contacts}")  
 return* contacts, *None  
  
 def* find\_value(*self*, key: ID, sender: Contact) \  
 -> tuple[list[Contact] | *None*, str | *None*]:  
 *"""  
 Sends key values if new contact, then attempts to find the value of a key-value pair in  
 our storage (then cache storage), given the key. If it cannot do that, it will return  
 K contacts that are closer to the key than it is.  
 """  
 if* sender.id == *self*.our\_contact.id:  
 *raise* SendingQueryToSelfError("Sender cannot be ourselves.")  
  
 *self*.send\_key\_values\_if\_new\_contact(sender)  
  
 *if self*.storage.contains(key):  
 print(f"[DEBUG] Value in self.storage of {*self*.our\_contact.id}.")  
 *return None*, *self*.storage.get(key)  
 *elif self*.cache\_storage.contains(key):  
 print(f"[DEBUG] Value in self.cache\_storage of {*self*.our\_contact.id}.")  
 *return None*, *self*.cache\_storage.get(key)  
 *else*:  
 print("[DEBUG] Value not in storage, getting close contacts.")  
 *return self*.bucket\_list.get\_close\_contacts(key, sender.id), *None  
  
 def* send\_key\_values\_if\_new\_contact(*self*, sender: Contact) -> *None*:  
 *"""  
 Spec: "When a new node joins the system, it must store any  
 key-value pair to which it is one of the k closest. Existing  
 nodes, by similarly exploiting complete knowledge of their  
 surrounding subtrees, will know which key-value pairs the new  
 node should store. Any node learning of a new node therefore  
 issues STORE RPCs to transfer relevant key-value pairs to the  
 new node. To avoid redundant STORE RPCs, however, a node only  
 transfers a key-value pair if its own ID is closer to the key  
 than are the IDs of other nodes."  
  
 For a new contact, we store values to that contact whose keys  
 XOR our\_contact are less than the stored keys XOR other\_contacts.  
 """  
 # print("send key values if new contact")  
 if self*.\_is\_new\_contact(sender):  
 *# with self.bucket\_list.lock:  
 # Clone, so we can release the lock.* contacts: list[Contact] = *self*.bucket\_list.contacts()  
 *if* len(contacts) > 0:  
 *# and our distance to the key < any other contact's distance  
 # to the key  
 for* k *in self*.storage.get\_keys():  
 *# our minimum distance to the contact.* distance = min([c.id ^ k *for* c *in* contacts])  
 *# If our contact is closer, store the contact on its  
 # node.  
 if* (*self*.our\_contact.id ^ k) < distance:  
 print(sender.protocol)  
 error: RPCError | *None* = sender.protocol.store(  
 sender=*self*.our\_contact,  
 key=ID(k),  
 val=*self*.storage.get(k)  
 )  
 *if self*.dht:  
 *self*.dht.handle\_error(error, sender)  
  
 *def* \_is\_new\_contact(*self*, sender: Contact) -> bool:  
 *"""  
 Returns NOT(if the contact exists in our bucket list or in our DHT’s pending contact list.)  
 :param sender:  
 :return:  
 """* ret: bool  
 *# with self.bucket\_list.lock:* ret: bool = *self*.bucket\_list.contact\_exists(sender)  
 *# end lock  
 if self*.dht: *# might be None in unit testing  
 # with self.DHT.pending\_contacts.lock:* ret |= (sender.id *in* [c.id *for* c *in self*.dht.pending\_contacts])  
 *# end lock  
  
 return not* ret  
  
 *def* simply\_store(*self*, key, val) -> *None*:  
 *"""  
 For unit testing.  
 :param key:  
 :param val:  
 :return: None  
 """  
 self*.storage.set(key, val)  
  
 *# Server entry points  
  
 def* server\_ping(*self*, request: CommonRequest) -> dict:  
 print("[Server] Ping called")  
 protocol: IProtocol = request["protocol"]  
 *self*.ping(  
 Contact(  
 protocol=protocol,  
 id=ID(request["sender"])  
 )  
 )  
 *return* {"random\_id": request["random\_id"]}  
  
 *def* server\_store(*self*, request: CommonRequest) -> dict:  
 print("[Server] Server store called.")  
 protocol: IProtocol = request["protocol"]  
 *self*.store(  
 sender=Contact(  
 id=ID(request["sender"]),  
 protocol=protocol  
 ),  
 key=ID(request["key"]),  
 val=str(request["value"]),  
 is\_cached=request["is\_cached"],  
 expiration\_time\_sec=request["expiration\_time\_sec"]  
 )  
 *return* {"random\_id": request["random\_id"]}  
  
 *def* server\_find\_node(*self*, request: CommonRequest) -> dict:  
 print("[Server] Find node called")  
 protocol: IProtocol = request["protocol"]  
  
 contacts, val = *self*.find\_node(  
 sender=Contact(  
 protocol=protocol,  
 id=ID(request["sender"])  
 ),  
 key=ID(request["key"])  
 )  
  
 contact\_dict: list[dict] = []  
 *for* c *in* contacts:  
 contact\_info = {  
 "contact": c.id.value,  
 "protocol": c.protocol,  
 "protocol\_name": type(c.protocol)  
 }  
  
 contact\_dict.append(contact\_info)  
  
 *return* {"contacts": contact\_dict, "random\_id": request["random\_id"]}  
  
 *def* server\_find\_value(*self*, request: CommonRequest) -> dict:  
 print("[Server] Find Value called")  
 protocol: IProtocol = request["protocol"]  
 print(protocol)  
 contacts, val = *self*.find\_value(  
 sender=Contact(  
 protocol=protocol,  
 id=ID(request["sender"])  
 ),  
 key=ID(request["key"])  
 )  
 print(contacts, val)  
 contact\_dict: list[dict] = []  
 *if* contacts:  
 *for* c *in* contacts:  
 contact\_info = {  
 "contact": c.id.value,  
 "protocol": c.protocol,  
 "protocol\_name": type(c.protocol)  
 }  
 contact\_dict.append(contact\_info)  
 *return* {"contacts": contact\_dict,  
 "random\_id": request["random\_id"],  
 "value": val}

### pickler.py

*import* pickle  
  
*from* kademlia.errors *import* DataDecodingError  
  
  
*def* encode\_data(data: dict) -> bytes:  
 *"""  
 Takes in a dictionary, encodes all values using pickle, in order to retain objects  
 over HTTP.  
 The dictionary is then converted to a string using json.dumps()  
 """  
 return* pickle.dumps(data)  
  
  
*def* plain\_encode\_data(data: dict) -> bytes:  
 *"""  
 Takes in a dictionary, encodes all values using pickle, in order to retain objects  
 over HTTP.  
 The dictionary is then converted to a string using json.dumps()  
 """  
 return* pickle.dumps(data)  
  
  
*def* decode\_data(encoded\_data: bytes) -> dict:  
 *"""  
 Takes in a string, decodes all pickled byte strings of the string dictionary   
 into python objects, and returns the decoded dictionary.  
 """  
 try*:  
 *if* isinstance(encoded\_data, bytes):  
 decoded\_data = pickle.loads(encoded\_data)  
 *else*:  
 *raise* TypeError(f"Encoded data should be type bytes, found type {type(encoded\_data)}")  
  
 *except* Exception *as* error:  
 *raise* DataDecodingError("Error decoding data.") *from* error  
 *return* decoded\_data  
  
  
*def* plain\_decode\_data(encoded\_data: bytes) -> dict:  
 *"""  
 Takes in a string, decodes all pickled byte strings of the string dictionary  
 into python objects, and returns the decoded dictionary.  
 """  
 try*:  
 *if* isinstance(encoded\_data, bytes):  
 decoded\_data = pickle.loads(encoded\_data)  
 *else*:  
 *raise* TypeError(f"Encoded data should be type bytes, found type {type(encoded\_data)}")  
  
 *except* Exception *as* error:  
 *raise* DataDecodingError("Error decoding data.") *from* error  
 *return* decoded\_data  
  
  
*if* \_\_name\_\_ == "\_\_main\_\_":  
   
 *class* MyClass:  
 *def* \_\_init\_\_(*self*, defined):  
 *self*.static\_attr = "static"  
 *self*.defined\_attr = defined  
 *self*.\_protected\_attr = "protected"  
 *self*.\_\_private\_attr = "private"  
  
 *def* method(*self*):  
 *return self*.\_\_private\_attr, *self*.defined\_attr  
  
 my\_dict = {"a": 1, "b": 27, "c": [1, 2, 3, MyClass("defined in dict")]}  
 print(my\_dict)  
 enc = encode\_data(my\_dict)  
 dec = decode\_data(enc)  
 print(dec)  
 print(dec["c"][3].method())

### protocols.py

*from* typing *import* Optional  
  
*import* requests  
  
*from* kademlia *import* pickler  
*from* kademlia.constants *import* Constants  
*from* kademlia.contact *import* Contact  
*from* kademlia.dictionaries *import* (BaseResponse, ErrorResponse, FindNodeSubnetRequest,  
 FindValueSubnetRequest, PingSubnetRequest, StoreSubnetRequest, FindNodeRequest,  
 FindValueRequest, PingRequest, StoreRequest)  
*from* kademlia.errors *import* RPCError  
*from* kademlia.id *import* ID  
*from* kademlia.interfaces *import* IProtocol  
*from* kademlia.node *import* Node  
*from* kademlia.pickler *import* encode\_data  
  
  
*def* get\_rpc\_error(id: ID,  
 ret: BaseResponse | *None*,  
 timeout\_error: bool,  
 peer\_error: ErrorResponse) -> RPCError:  
 *# print("Peer error:", peer\_error)* error = RPCError()  
 *if* ret:  
 error.id\_mismatch\_error = id != ret["random\_id"]  
 *else*:  
 error.id\_mismatch\_error = *False* error.timeout\_error = timeout\_error  
 error.peer\_error = peer\_error["error\_message"] *not in* ["", *None*]  
 *if* peer\_error["error\_message"]:  
 error.peer\_error\_message = peer\_error["error\_message"]  
  
 *return* error  
  
  
*class* VirtualProtocol(IProtocol):  
 *"""  
 For unit testing, doesn't really do much in the main  
 implementation, it's just used to make sure everything that  
 doesn't involve networking works correctly.  
 """  
  
 def* \_\_init\_\_(*self*, node: Node | *None* = *None*, responds=*True*) -> *None*:  
 *self*.responds = responds  
 *self*.node = node  
 *self*.type = "VirtualProtocol"  
  
 *def* ping(*self*, sender: Contact) -> RPCError:  
 *"""  
 Pings sender if we respond.  
  
 :param sender:  
 :return:  
 """  
 if self*.responds:  
 *self*.node.ping(sender)  
 *return* RPCError.no\_error()  
 *else*:  
 error = RPCError(  
 "Time out while pinging contact - VirtualProtocol does not respond.",  
 timeout\_error=*not self*.responds  
 )  
 *return* error  
  
 *def* find\_node(*self*, sender: Contact,  
 key: ID) -> tuple[list[Contact], RPCError]:  
 *"""  
 Finds K close contacts to a given ID, while excluding the sender.  
 It also adds the sender if it hasn't seen it before.  
 :param key: K close contacts are found near this ID.  
 :param sender: Contact to be excluded and added if new.  
 :return: list of K (or less) contacts near the key, and an error that may need to be handled.  
 """  
 return self*.node.find\_node(sender=sender, key=key)[0], RPCError.no\_error()  
  
 *def* find\_value(*self*, sender: Contact,  
 key: ID) -> tuple[list[Contact] | *None*, str | *None*, RPCError]:  
 *"""  
 Sends key values if new contact, then attempts to find the value of a key-value pair in  
 our storage (then cache storage), given the key. If it cannot do that, it will return  
 K contacts that are closer to the key than it is.  
 """* contacts, val = *self*.node.find\_value(sender=sender, key=key)  
 *return* contacts, val, RPCError.no\_error()  
  
 *def* store(*self*,  
 sender: Contact,  
 key: ID,  
 val: str,  
 is\_cached=*False*,  
 exp\_time\_sec: int = 0) -> RPCError:  
 *"""  
 Stores the key-value on the remote peer.  
 """  
 self*.node.store(sender=sender,  
 key=key,  
 val=val,  
 is\_cached=is\_cached,  
 expiration\_time\_sec=exp\_time\_sec)  
  
 *return* RPCError.no\_error()  
  
  
*class* TCPSubnetProtocol(IProtocol):  
  
 *def* \_\_init\_\_(*self*, url: str, port: int, subnet: int):  
 *self*.url = url  
 *self*.port = port  
 *self*.responds = *True  
 self*.subnet = subnet  
 *self*.type = "TCPSubnetProtocol"  
  
 *def* find\_node(*self*, sender: Contact, key: ID) -> tuple[list[Contact] | *None*, RPCError]:  
 *"""  
 Encodes all of the data that is needed into a FindNodeSubnetRequest,  
 Which is then pickled and posted using the ‘requests’ library to self.url and  
 self.port using a “/find\_node” endpoint. This makes sense because our node doesn’t  
 call this – other nodes will call this method to contact us. This handles a timeout error  
 by creating a timeout error RPCError, any other errors are also turned into and RPCError by  
 get\_rpc\_error().  
  
 :param sender:  
 :param key:  
 :return:  
 """* id: ID = ID.random\_id()  
  
 encoded\_data = encode\_data(  
 dict(FindNodeSubnetRequest(  
 protocol=sender.protocol,  
 protocol\_name=type(sender.protocol),  
 subnet=*self*.subnet,  
 sender=sender.id.value,  
 key=key.value,  
 random\_id=id.value  
 ))  
 )  
 print(f"http://{*self*.url}:{*self*.port}/find\_node")  
  
 ret = *None* timeout\_error = *False* error = ""  
 *try*:  
 print("[Client] Sending find\_node RPC...")  
 ret = requests.post(  
 f"http://{*self*.url}:{*self*.port}/find\_node",  
 data=encoded\_data,  
 timeout=Constants.REQUEST\_TIMEOUT  
 )  
 print(ret)  
  
 *except* requests.Timeout *as* t:  
 print("[ERROR] [Client] Timeout error when contacting node.\n", t)  
 timeout\_error = *True* error = t  
  
 *except* Exception *as* e:  
 print("[ERROR] [Client]", e)  
 *# request timed out.* timeout\_error = *False* error = e  
  
 *if* ret:  
 encoded\_data = ret.content  
 ret\_decoded = pickler.decode\_data(encoded\_data)  
 *else*:  
 ret\_decoded = *None  
 try*:  
 *if* ret\_decoded:  
 *if* ret\_decoded["contacts"]:  
 contacts = []  
 *for* val *in* ret\_decoded["contacts"]:  
 new\_c = Contact(ID(val["contact"]), val["protocol"])  
 contacts.append(new\_c)  
 *# Return only contacts with supported protocols.* rpc\_error = get\_rpc\_error(id,  
 ret\_decoded,  
 timeout\_error,  
 ErrorResponse(error\_message=str(error), random\_id=ID.random\_id()))  
 *if* contacts:  
 ret\_contacts = [c *for* c *in* contacts *if* c.protocol *is not None*]  
 *return* ret\_contacts, rpc\_error  
 *else*:  
 rpc\_error = get\_rpc\_error(id,  
 ret\_decoded,  
 timeout\_error,  
 ErrorResponse(error\_message=str(error), random\_id=ID.random\_id()))  
 *return* [], rpc\_error  
 *except* Exception *as* e:  
 error = RPCError()  
 error.protocol\_error = *True* print("[Client] Exception thrown: ", e)  
 *return None*, error  
  
 *def* find\_value(*self*, sender: Contact, key: ID) -> tuple[list[Contact] | *None*, str | *None*, RPCError | *None*]:  
 *"""  
 Attempt to find the value in the peer network.  
  
 A null contact list is acceptable as it is a valid return  
 if the value is found.  
 The caller is responsible for checking the timeoutError flag  
 to make sure null contacts is not the result of a timeout  
 error.  
  
 Encodes all the data that is needed into a FindValueSubnetRequest,  
 Which is then pickled and posted using the ‘requests’ library to self.url  
 and self.port using a “/find\_value” endpoint. This makes sense because our  
 node doesn’t call this – other nodes will call this method to contact us.  
 This handles a timeout error by creating a timeout error RPCError, any other  
 errors are also turned into and RPCError by get\_rpc\_error().  
  
  
 :param sender: Sender to find value from  
 :param key: Key to check for value from key-value pair  
 :return: contacts, value, RPCError  
 """* random\_id = ID.random\_id()  
 encoded\_data = encode\_data(  
 dict(FindValueSubnetRequest(  
 protocol=sender.protocol,  
 protocol\_name=type(sender.protocol),  
 subnet=*self*.subnet,  
 sender=sender.id.value,  
 key=key.value,  
 random\_id=random\_id.value  
 ))  
 )  
  
 ret = *None  
 try*:  
 print("[Client] Sending POST")  
 ret = requests.post(  
 url=f"http://{*self*.url}:{*self*.port}/find\_value",  
 data=encoded\_data,  
 timeout=Constants.REQUEST\_TIMEOUT  
 )  
 print("[Client] Completed POST")  
 timeout\_error = *False* error = *None  
  
 except* requests.Timeout *as* t:  
 print("Timeout error", t)  
 timeout\_error = *True* error = t  
  
 *except* Exception *as* e:  
 print("Exception!", e)  
 *# request timed out.* timeout\_error = *False* error = e  
  
 ret\_decoded = *None  
 if* ret:  
 encoded\_data = ret.content  
 ret\_decoded = pickler.decode\_data(encoded\_data)  
  
 *try*:  
 contacts = []  
 *if* ret\_decoded:  
 *if* ret\_decoded["contacts"]:  
 *for* c *in* ret\_decoded["contacts"]:  
 new\_contact = Contact(  
 c["protocol"], *# instantiate\_protocol* ID(c["contact"])  
 )  
 contacts.append(new\_contact)  
 print("about to return")  
  
 *return* [c *for* c *in* contacts *if* c.protocol *is not None*], \  
 ret\_decoded["value"], \  
 get\_rpc\_error(  
 random\_id, ret\_decoded, timeout\_error, ErrorResponse(  
 random\_id=random\_id.value,  
 error\_message=str(error))  
 )  
 *else*:  
 *return* [c *for* c *in* contacts *if* c.protocol *is not None*], "", get\_rpc\_error(  
 random\_id, ret\_decoded, timeout\_error, ErrorResponse(  
 random\_id=random\_id.value,  
 error\_message=str(error))  
 )  
 *except* Exception *as* e:  
 rpc\_error = RPCError(str(e))  
 rpc\_error.protocol\_error = *True* print(f"[Client] Error performing find\_value: {rpc\_error}")  
 *return None*, *None*, rpc\_error  
  
 *def* ping(*self*, sender: Contact) -> RPCError:  
 *"""  
 Encodes all of the data that is needed into a PingSubnetRequest,  
 Which is then pickled and posted using the ‘requests’ library to self.url and  
 self.port using a “/ping” endpoint. This makes sense because our node doesn’t call  
 this – other nodes will call this method to contact us. This handles a timeout error  
 by setting a timeout\_error flag, which is passed into get\_rpc\_error at the end of  
 the method. Any other exceptions are handled in a similar fashion.  
  
 The response is then decoded if there is one, then an RPCError is returned.  
  
 :param sender:  
 :return:  
 """* random\_id = ID.random\_id()  
 encoded\_data = encode\_data(  
 dict(PingSubnetRequest(  
 protocol=sender.protocol,  
 protocol\_name=type(sender.protocol),  
 subnet=*self*.subnet,  
 sender=sender.id.value,  
 random\_id=random\_id.value)))  
  
 timeout\_error = *False* error = *None* ret = *None  
 try*:  
 print("[Client] Sending Ping RPC...")  
 ret: requests.Response = requests.post(  
 url=f"http://{*self*.url}:{*self*.port}/ping",  
 data=encoded\_data,  
 timeout=Constants.REQUEST\_TIMEOUT  
 )  
 print(f"[Client] Received HTTP Response from {ret.url} with code {ret.status\_code}")  
  
 *except* requests.Timeout *as* t:  
 print("[Client] Ping timeout error: ", t)  
 timeout\_error = *True* error = t  
  
 *except* Exception *as* e:  
 print("[ERROR] [Client] Other exception thrown (Ping): ", e)  
 *# request timed out.* timeout\_error = *False* error = e  
  
 ret\_base\_response = *None* formatted\_response = *None  
 if* ret:  
 encoded\_data = ret.content  
 formatted\_response = pickler.decode\_data(encoded\_data)  
  
 *return* get\_rpc\_error(random\_id, formatted\_response, timeout\_error, ErrorResponse(  
 error\_message=str(error), random\_id=ID.random\_id()))  
  
 *def* store(*self*,  
 sender: Contact,  
 key: ID,  
 val: str,  
 is\_cached=*False*,  
 expiration\_time\_sec=0  
 ) -> RPCError:  
 random\_id = ID.random\_id()  
  
 encoded\_data = encode\_data(  
 dict(StoreSubnetRequest(  
 protocol=sender.protocol,  
 protocol\_name=type(sender.protocol),  
 subnet=*self*.subnet,  
 sender=sender.id.value,  
 key=key.value,  
 value=val,  
 is\_cached=is\_cached,  
 expiration\_time\_sec=expiration\_time\_sec,  
 random\_id=random\_id.value)))  
  
 timeout\_error = *False* error = *None* ret = *None  
  
 try*:  
 print(f"[Client] Running Store POST to http://{*self*.url}:{*self*.port}/store")  
 ret = requests.post(  
 url=f"http://{*self*.url}:{*self*.port}/store",  
 data=encoded\_data,  
 timeout=Constants.REQUEST\_TIMEOUT  
 )  
 print("[Client] Store POST done!")  
  
 *except* requests.Timeout *as* t:  
 print("[Client] Timeout error when contacting node.")  
 timeout\_error = *True* error = t  
  
 *except* Exception *as* e:  
 print("Exception!", e)  
 *# request timed out.* timeout\_error = *False* error = e  
  
 formatted\_response = *None  
 if* ret:  
 encoded\_data = ret.content  
 formatted\_response = pickler.decode\_data(encoded\_data)  
  
 *return* get\_rpc\_error(random\_id, formatted\_response, timeout\_error, ErrorResponse(  
 error\_message=str(error), random\_id=ID.random\_id()))  
  
  
*class* TCPProtocol(IProtocol):  
  
 *def* \_\_init\_\_(*self*, url: str, port: int):  
 *self*.url = url  
 *self*.port = port  
 *self*.responds = *True  
 self*.type = "TCPProtocol"  
  
 *def* find\_node(*self*, sender: Contact, key: ID) -> tuple[list[Contact] | *None*, RPCError]:  
 id: ID = ID.random\_id()  
 encoded\_data = encode\_data(  
 dict(FindNodeRequest(  
 protocol=sender.protocol,  
 protocol\_name=type(sender.protocol),  
 sender=sender.id.value,  
 key=key.value,  
 random\_id=id.value  
 ))  
 )  
 *# print(f"http://{self.url}:{self.port}/find\_node")* ret = *None* timeout\_error = *False* error = ""  
 *try*:  
 print("[Client] Sending find\_node RPC...")  
 ret = requests.post(  
 f"http://{*self*.url}:{*self*.port}/find\_node",  
 data=encoded\_data,  
 timeout=Constants.REQUEST\_TIMEOUT  
 )  
 print(ret)  
  
 *except* requests.Timeout *as* t:  
 print("[ERROR] [Client] Timeout error when contacting node.\n", t)  
 timeout\_error = *True* error = t  
  
 *except* Exception *as* e:  
 print("[ERROR] [Client]", e)  
 *# request timed out.* timeout\_error = *False* error = e  
 *if* ret:  
 encoded\_data = ret.content  
 ret\_decoded = pickler.decode\_data(encoded\_data)  
 *else*:  
 ret\_decoded = *None  
 try*:  
 *if* ret\_decoded:  
 *if* ret\_decoded["contacts"]:  
 contacts = []  
 *for* val *in* ret\_decoded["contacts"]:  
 new\_c = Contact(ID(val["contact"]), val["protocol"])  
 contacts.append(new\_c)  
 *# Return only contacts with supported protocols.* rpc\_error = get\_rpc\_error(id,  
 ret\_decoded,  
 timeout\_error,  
 ErrorResponse(error\_message=str(error), random\_id=ID.random\_id()))  
 *if* contacts:  
 ret\_contacts = [c *for* c *in* contacts *if* c.protocol *is not None*]  
 *return* ret\_contacts, rpc\_error  
 rpc\_error = get\_rpc\_error(id,  
 ret\_decoded,  
 timeout\_error,  
 ErrorResponse(error\_message=str(error), random\_id=ID.random\_id()))  
 *return* [], rpc\_error  
 *except* Exception *as* e:  
 error = RPCError()  
 error.protocol\_error = *True* print("[Client] Exception thrown: ", e)  
 *return None*, error  
  
 *def* find\_value(*self*, sender: Contact, key: ID) -> tuple[list[Contact] | *None*, str | *None*, RPCError | *None*]:  
 *"""  
 Attempt to find the value in the peer network.  
  
 A null contact list is acceptable as it is a valid return  
 if the value is found.  
 The caller is responsible for checking the timeoutError flag  
 to make sure null contacts is not the result of a timeout  
 error.  
  
 :param sender: Sender to find value from  
 :param key: Key to check for value from key-value pair  
 :return: contacts, value, RPCError  
 """* random\_id = ID.random\_id()  
 encoded\_data = encode\_data(  
 dict(FindValueRequest(  
 protocol=sender.protocol,  
 protocol\_name=type(sender.protocol),  
 sender=sender.id.value,  
 key=key.value,  
 random\_id=random\_id.value  
 ))  
 )  
  
 ret = *None  
 try*:  
 print("[Client] Sending POST")  
 ret = requests.post(  
 url=f"http://{*self*.url}:{*self*.port}/find\_value",  
 data=encoded\_data,  
 timeout=Constants.REQUEST\_TIMEOUT  
 )  
 print("[Client] Completed POST")  
 timeout\_error = *False* error = *None  
  
 except* requests.Timeout *as* t:  
 print("Timeout error", t)  
 timeout\_error = *True* error = t  
  
 *except* Exception *as* e:  
 print("Exception!", e)  
 *# request timed out.* timeout\_error = *False* error = e  
  
 ret\_decoded = *None  
 if* ret:  
 encoded\_data = ret.content  
 ret\_decoded = pickler.decode\_data(encoded\_data)  
  
 *try*:  
 contacts = []  
 *if* ret\_decoded:  
 *if* ret\_decoded["contacts"]:  
 *for* c *in* ret\_decoded["contacts"]:  
 new\_contact = Contact(  
 c["protocol"], *# instantiate\_protocol* ID(c["contact"])  
 )  
 contacts.append(new\_contact)  
 print("about to return")  
  
 *return* [c *for* c *in* contacts *if* c.protocol *is not None*], \  
 ret\_decoded["value"], \  
 get\_rpc\_error(  
 random\_id, ret\_decoded, timeout\_error, ErrorResponse(  
 random\_id=random\_id.value,  
 error\_message=str(error))  
 )  
 *else*:  
 *return* [c *for* c *in* contacts *if* c.protocol *is not None*], "", get\_rpc\_error(  
 random\_id, ret\_decoded, timeout\_error, ErrorResponse(  
 random\_id=random\_id.value,  
 error\_message=str(error))  
 )  
 *except* Exception *as* e:  
 rpc\_error = RPCError(str(e))  
 rpc\_error.protocol\_error = *True* print(f"[Client] Error performing find\_value: {rpc\_error}")  
 *return None*, *None*, rpc\_error  
  
 *def* ping(*self*, sender: Contact) -> RPCError:  
 random\_id = ID.random\_id()  
 encoded\_data = encode\_data(  
 dict(PingRequest(  
 protocol=sender.protocol,  
 protocol\_name=type(sender.protocol),  
 sender=sender.id.value,  
 random\_id=random\_id.value)))  
  
 timeout\_error = *False* error = *None* ret: Optional[requests.Response] = *None  
 try*:  
 print("[Client] Sending Ping RPC...")  
 ret = requests.post(  
 url=f"http://{*self*.url}:{*self*.port}/ping",  
 data=encoded\_data,  
 timeout=Constants.REQUEST\_TIMEOUT  
 )  
 print(f"[Client] Received HTTP Response from {ret.url} with code {ret.status\_code}")  
  
 *except* requests.Timeout *as* t:  
 print("[Client] Ping timeout error: ", t)  
 timeout\_error = *True* error = t  
  
 *except* Exception *as* e:  
 print("[ERROR] [Client] Other exception thrown (Ping): ", e)  
 *# request timed out.* timeout\_error = *False* error = e  
  
 formatted\_response = *None  
 if* ret:  
 encoded\_data = ret.content  
 formatted\_response = pickler.decode\_data(encoded\_data)  
  
 *return* get\_rpc\_error(random\_id, formatted\_response, timeout\_error, ErrorResponse(  
 error\_message=str(error), random\_id=ID.random\_id()))  
  
 *def* store(*self*,  
 sender: Contact,  
 key: ID,  
 val: str,  
 is\_cached=*False*,  
 expiration\_time\_sec=0  
 ) -> RPCError:  
 random\_id = ID.random\_id()  
  
 encoded\_data = encode\_data(  
 dict(StoreRequest(  
 protocol=sender.protocol,  
 protocol\_name=type(sender.protocol),  
 sender=sender.id.value,  
 key=key.value,  
 value=val,  
 is\_cached=is\_cached,  
 expiration\_time\_sec=expiration\_time\_sec,  
 random\_id=random\_id.value)))  
  
 timeout\_error = *False* error = *None* ret = *None  
  
 try*:  
 print(f"[Client] Running Store POST to http://{*self*.url}:{*self*.port}/store")  
 ret = requests.post(  
 url=f"http://{*self*.url}:{*self*.port}/store",  
 data=encoded\_data,  
 timeout=Constants.REQUEST\_TIMEOUT  
 )  
 print("[Client] Store POST done!")  
  
 *except* requests.Timeout *as* t:  
 print("[Client] Timeout error when contacting node.")  
 timeout\_error = *True* error = t  
  
 *except* Exception *as* e:  
 print("Exception!", e)  
 *# request timed out.* timeout\_error = *False* error = e  
  
 formatted\_response = *None  
 if* ret:  
 encoded\_data = ret.content  
 formatted\_response = pickler.decode\_data(encoded\_data)  
  
 *return* get\_rpc\_error(random\_id, formatted\_response, timeout\_error, ErrorResponse(  
 error\_message=str(error), random\_id=ID.random\_id()))

### routers.py

*import* threading  
*from* abc *import* abstractmethod  
*from* datetime *import* datetime  
*from* time *import* sleep  
*from* typing *import* Callable, Optional  
  
*import* kademlia.my\_queues *as* my\_queues  
*from* kademlia.buckets *import* KBucket  
*from* kademlia.constants *import* Constants  
*from* kademlia.contact *import* Contact  
*from* kademlia.dictionaries *import* ContactQueueItem, FindResult  
*from* kademlia.errors *import* AllKBucketsAreEmptyError, ValueCannotBeNoneError  
*from* kademlia.id *import* ID  
*from* kademlia.node *import* Node  
  
  
*class* BaseRouter:  
 *def* \_\_init\_\_(*self*, node: Node):  
 *self*.closer\_contacts: list[Contact] = []  
 *self*.further\_contacts: list[Contact] = []  
 *self*.node: Node = node  
 *self*.dht = *None  
 # self.locker  
  
 def* find\_closest\_nonempty\_kbucket(*self*, key: ID) -> KBucket:  
 *"""  
 Finds the closest non empty Kbucket in our nodes bucket list to a given key.  
 :param key:  
 :return:  
 """  
 # gets all non-empty buckets from bucket list* non\_empty\_buckets: list[KBucket] = [  
 b *for* b *in self*.node.bucket\_list.buckets *if* (len(b.contacts) != 0)  
 ]  
 *if* len(non\_empty\_buckets) == 0:  
 *raise* AllKBucketsAreEmptyError(  
 "No non-empty buckets can be found.")  
  
 *return* sorted(non\_empty\_buckets,  
 key=(*lambda* b: b.id.value ^ key.value))[0]  
  
 *def* rpc\_find\_nodes(*self*, key: ID, contact: Contact):  
 *"""  
 Performs find nodes() on “contact”, where we are the sender, searching for “key”.  
 :param key:  
 :param contact:  
 :return:  
 """* new\_contacts, timeout\_error = contact.protocol.find\_node(  
 *self*.node.our\_contact, key)  
  
 *if self*.dht:  
 *self*.dht.handle\_error(timeout\_error, contact)  
  
 *return* new\_contacts, *None*, *None  
  
 def* rpc\_find\_value(*self*, key: ID, contact: Contact) -> tuple[list[Contact], Contact, str]:  
 *"""  
 Performs find value() on “contact”, where we are the sender searching for a value  
 corresponding to “key”, or K contacts close to “key”, if we find the value, we  
 will also receive the contact that contains it.  
 :param key:  
 :param contact:  
 :return:  
 """* nodes: list[Contact] = []  
 ret\_val: Optional[str] = *None* found\_by: Optional[Contact] = *None* other\_contacts, val, error = contact.protocol.find\_value(*self*.node.our\_contact, key)  
 *if self*.dht:  
 *self*.dht.handle\_error(error, contact)  
 *else*:  
 print("[Client] Router: No DHT to handle possible error.\nError:", error)  
  
 *if not* error *or not* error.has\_error():  
 *if* other\_contacts *is not None*:  
 *for* other\_contact *in* other\_contacts:  
 nodes.append(other\_contact)  
 *else*:  
 *if* val *is None*:  
 *raise* ValueCannotBeNoneError("None values are not expected, nor supported from FIND\_VALUE RPC.")  
 *else*:  
 nodes.append(contact)  
 found\_by = contact  
 ret\_val = val  
  
 *return* nodes, found\_by, ret\_val  
  
 *def* \_query(*self*,  
 key: ID,  
 nodes\_to\_query: list[Contact],  
 rpc\_call: Callable,  
 closer\_contacts: list[Contact],  
 further\_contacts: list[Contact]) -> FindResult:  
 *"""  
 Gets nodes that are closer to “key” than a node we are querying – we query all of nodes\_to\_query.  
 This ends as soon as we find a value. A FindResult object is then returned containing closer\_contacts,  
 found, found\_by, and the value we found.  
  
 Gets nodes by performing rpc-call on the node to query, looking for “key”, if we don’t know it,  
 it is added to a list “peers\_nodes”. Checking each contact in peers\_nodes, we check if it is closer  
 to the key than the node\_to\_query, if it is, we add it to closer\_contacts. We do a similar check to  
 check if it is further, and then we add it to further\_contacts.  
  
 :param key:  
 :param nodes\_to\_query:  
 :param rpc\_call:  
 :param closer\_contacts:  
 :param further\_contacts:  
 :return:  
 """* found: bool = *False* found\_by: Optional[Contact] = *None* val: str = ""  
  
 *for* n *in* nodes\_to\_query:  
 found, val, found\_by, closer\_contacts, further\_contacts = *self*.get\_closer\_nodes(  
 key=key,  
 node\_to\_query=n,  
 rpc\_call=rpc\_call,  
 closer\_contacts=closer\_contacts,  
 further\_contacts=further\_contacts  
 )  
 *if* found:  
 *break  
  
 return* FindResult(  
 found=found,  
 contacts=closer\_contacts,  
 found\_by=found\_by,  
 val=val  
 )  
  
 @abstractmethod  
 *def* lookup(*self*, key: ID, rpc\_call: Callable, give\_me\_all=*False*) -> FindResult | *None*:  
 *pass* @staticmethod  
 *def* get\_closest\_nodes(key: ID, bucket: KBucket) -> list[Contact]:  
 *"""  
 Get sorted list of closest contacts to the given key.  
 :param key: key to look close to.  
 :param bucket: bucket to look in.  
 :return: sorted list of contacts by distance (sorted by XOR distance to parameter key)  
 """  
 return* sorted(bucket.contacts, key=*lambda* c: c.id ^ key)  
  
 *def* get\_closer\_nodes(*self*,  
 key: ID,  
 node\_to\_query: Contact,  
 rpc\_call: Callable[[ID, Contact], tuple[list[Contact], Contact, str]],  
 further\_contacts: list[Contact],  
 closer\_contacts: list[Contact]  
 ) -> tuple[bool, str, Contact, list[Contact], list[Contact]]:  
 *"""  
 Gets nodes that are closer to “key” than “node\_to\_query”.  
  
 Gets nodes by performing rpc-call on the node to query, looking for “key”, if we don’t know it,  
 it is added to a list “peers\_nodes”. Checking each contact in peers\_nodes, we check if it is closer  
 to the key than the node\_to\_query, if it is, we add it to closer\_contacts. We do a similar check to  
 check if it is further, and then we add it to further\_contacts.  
  
 :param key:  
 :param node\_to\_query:  
 :param rpc\_call:  
 :param further\_contacts:  
 :param closer\_contacts:  
 :return:  
 """* contacts, found\_by, val = rpc\_call(key, node\_to\_query)  
 peers\_nodes: list[Contact] = []  
 *for* contact *in* contacts:  
 *if* contact.id.value *not in* [*self*.node.our\_contact.id.value, node\_to\_query.id.value]:  
 *if* contact *not in* [closer\_contacts, further\_contacts]:  
 peers\_nodes.append(contact)  
  
 nearest\_node\_distance = node\_to\_query.id ^ key  
  
 *# lock (locker)* close\_peer\_nodes = [p *for* p *in* peers\_nodes *if* (p.id ^ node\_to\_query.id) < nearest\_node\_distance]  
 *for* p *in* close\_peer\_nodes:  
 *if* p.id *not in* [c.id *for* c *in* closer\_contacts]:  
 closer\_contacts.append(p)  
  
 *# lock (locker)* far\_peer\_nodes = [p *for* p *in* peers\_nodes *if* (p.id ^ node\_to\_query.id) >= nearest\_node\_distance]  
 *for* p *in* far\_peer\_nodes:  
 *if* p.id *not in* [c.id *for* c *in* further\_contacts]:  
 further\_contacts.append(p)  
  
 *return* val *is not None*, val, found\_by, closer\_contacts, further\_contacts  
  
  
*class* Router(BaseRouter):  
  
 *def* \_\_init\_\_(*self*, node: Node = *None*) -> *None*:  
 super().\_\_init\_\_(node)  
 *# self.lock = WithLock(Lock())  
  
 def* lookup(*self*,  
 key: ID,  
 rpc\_call: Callable,  
 give\_me\_all: bool = *False*) -> FindResult:  
 *"""  
 This performs the main Kademlia lookup algorithm serially.  
 This method initiates a Kademlia lookup operation, searching for nodes closest to the given key.  
 starts by getting nodes from the closest non-empty k-bucket and continues querying nodes  
 serially until it has gathered responses from the closest k nodes or until we run out of  
 nodes to contact.  
  
  
  
 :param key: Key to be looked up  
 :param rpc\_call: RPC call to be used.  
 :param give\_me\_all: If all contacts should be returned or not - for testing purposes mainly.  
 :return: returns query result.  
 """* contacted\_nodes = []  
 *if* Constants.TRY\_CLOSEST\_BUCKET:  
 *# Spec: The lookup initator starts by picking a nodes from its closest non-empty k-bucket* bucket: KBucket = *self*.find\_closest\_nonempty\_kbucket(key)  
  
 *# Not in spec: sort by the closest nodes in the closest bucket.* all\_nodes: list[Contact] = *self*.node.bucket\_list.get\_close\_contacts(  
 key, *self*.node.our\_contact.id)[0:Constants.K]  
 nodes\_to\_query: list[Contact] = all\_nodes[0:Constants.A]  
  
 *for* i *in* all\_nodes[Constants.A + 1:]:  
 *self*.further\_contacts.append(i)  
 *else*:  
 *if* Constants.DEBUG:  
 all\_nodes: list[Contact] = *self*.node.bucket\_list.get\_kbucket(key).contacts[0:Constants.K]  
 *else*:  
 *# This is a bad way to get a list of close contacts with virtual nodes because we're always going to  
 # get the closest nodes right at the get go.* all\_nodes: list[Contact] = *self*.node.bucket\_list.get\_close\_contacts(  
 key, *self*.node.our\_contact.id)[0:Constants.K]  
 nodes\_to\_query: list[Contact] = all\_nodes[:Constants.A]  
  
 *# Also not explicitly in spec:  
 # Any closer node in the alpha list is immediately added to our closer contact list  
 # and any further node in the alpha list is immediately added to our further contact list.  
 for* n *in* nodes\_to\_query:  
 *if* (n.id ^ key) < (*self*.node.our\_contact.id ^ key):  
 *self*.closer\_contacts.append(n)  
 *else*:  
 *self*.further\_contacts.append(n)  
  
 *# The remaining contacts not tested yet can be put here.  
 for* n *in* all\_nodes[Constants.A + 1:]:  
 *self*.further\_contacts.append(n)  
  
 *# We're about to contact these nodes.  
 for* n *in* nodes\_to\_query:  
 *if* n.id *not in* [i.id *for* i *in* contacted\_nodes]:  
 contacted\_nodes.append(n)  
  
 *# Spec: The initiator then sends parallel, async FIND\_NODE RPCs to the "a" nodes it has chosen,  
 # "a" is a system-wide parameter, such as 3.* query\_result: FindResult = *self*.\_query(key, nodes\_to\_query, rpc\_call, *self*.closer\_contacts,  
 *self*.further\_contacts)  
 *if* query\_result["found"]:  
 *# For unit testing* closer\_contacts\_unittest = *self*.closer\_contacts  
 further\_contacts\_unittest = *self*.further\_contacts  
 *return* query\_result  
  
 *# Add any new closer contacts to the list we're going to return.* ret: list[Contact] = []  
 *for* c *in self*.closer\_contacts:  
 *if* c.id *not in* [i.id *for* i *in* ret]:  
 ret.append(c)  
  
 *# Spec: The lookup terminates when the initator has queried and received responses from the k closest nodes  
 # it has seen.* have\_work = *True  
 while* len(ret) < Constants.K *and* have\_work:  
 closer\_uncontacted\_nodes = [  
 i *for* i *in self*.closer\_contacts *if* i *not in* contacted\_nodes  
 ]  
 further\_uncontacted\_nodes = [  
 i *for* i *in self*.further\_contacts *if* i *not in* contacted\_nodes  
 ]  
  
 *# If we have uncontacted nodes, we still have work to be done.* have\_closer: bool = len(closer\_uncontacted\_nodes) > 0  
 have\_further: bool = len(further\_uncontacted\_nodes) > 0  
 have\_work: bool = have\_closer *or* have\_further  
  
 *# Spec: of the k nodes the initiator has heard of closest to the target,  
 # it picks the 'a' that it has not yet queried and resends the FIND\_NODE RPC to them.  
 if* have\_closer:  
 new\_nodes\_to\_query = closer\_uncontacted\_nodes[:Constants.A]  
 *for* c *in* new\_nodes\_to\_query:  
 *if* c.id *not in* [i.id *for* i *in* contacted\_nodes]:  
 contacted\_nodes.append(c)  
  
 query\_result = (*self*.\_query(key, new\_nodes\_to\_query, rpc\_call,  
 *self*.closer\_contacts,  
 *self*.further\_contacts))  
  
 *if* query\_result["found"]:  
 *# # For unit testing.  
 # closer\_contacts\_unittest = self.closer\_contacts  
 # further\_contacts\_unittest = self.further\_contacts  
 return* query\_result  
  
 *elif* have\_further:  
 new\_nodes\_to\_query = further\_uncontacted\_nodes[:Constants.A]  
 *for* c *in* further\_uncontacted\_nodes:  
 *if* c *not in* [i.id *for* i *in* contacted\_nodes]:  
 contacted\_nodes.append(c)  
  
 query\_result = (*self*.\_query(key, new\_nodes\_to\_query, rpc\_call,  
 *self*.closer\_contacts,  
 *self*.further\_contacts))  
  
 *if* query\_result["found"]:  
 *# # For unit testing.  
 # closer\_contacts\_unittest = self.closer\_contacts  
 # further\_contacts\_unittest = self.further\_contacts  
 return* query\_result  
  
 *# if DEBUG: # For unit testing  
 # closer\_contacts\_unittest = self.closer\_contacts  
 # further\_contacts\_unittest = self.further\_contacts  
  
 # return k closer nodes sorted by distance,  
  
 # Spec (sort of): return max(k) closer nodes, sorted by distance.  
 # For unit testing give\_me\_all can be true so that we can match against our alternate way of  
 # getting closer contacts.  
 # contacts, val, found, found\_by  
 return* FindResult(  
 found=*False*,  
 contacts=(ret *if* give\_me\_all *else* sorted(ret, key=*lambda* c: c.id ^ key)[:Constants.K]),  
 found\_by=*None*,  
 val=*None* )  
  
  
*class* ParallelRouter(BaseRouter):  
 *def* \_\_init\_\_(*self*, node: Node = *None*):  
 super().\_\_init\_\_(node)  
 *self*.\_\_contact\_queue = my\_queues.InfiniteLinearQueue()  
 *self*.\_\_semaphore = threading.Semaphore()  
 *self*.\_\_now: datetime = datetime.now()  
 *self*.\_\_stop\_work = *False  
 self*.\_\_threads: list[threading.Thread] = []  
 *self*.\_\_initialise\_thread\_pool()  
  
 *def* \_\_initialise\_thread\_pool(*self*) -> *None*:  
 *"""  
 Creates and starts MAX-THREADS # of threads, which all run self.rpc-caller.  
 :return:  
 """  
 for* \_ *in* range(Constants.MAX\_THREADS):  
 thread = threading.Thread(target=*self*.\_\_rpc\_caller)  
 *# thread.is\_background = True  
 self*.\_\_threads.append(thread)  
 thread.start()  
  
 *def* queue\_work(*self*,  
 key: ID,  
 contact: Contact,  
 rpc\_call: Callable,  
 closer\_contacts: list[Contact],  
 further\_contacts: list[Contact],  
 find\_result: FindResult) -> *None*:  
 *"""  
 Adds new Contact Queue Item to self.\_\_contact-queue, all the  
 parameters listed are added. The semaphore is released at the end of  
 this function to signal that there is work available in the queue.  
 :param key:  
 :param contact:  
 :param rpc\_call:  
 :param closer\_contacts:  
 :param further\_contacts:  
 :param find\_result:  
 :return:  
 """  
  
 self*.\_\_contact\_queue.enqueue(  
 ContactQueueItem(  
 key=key,  
 contact=contact,  
 rpc\_call=rpc\_call,  
 closer\_contacts=closer\_contacts,  
 further\_contacts=further\_contacts,  
 find\_result=find\_result)  
 )  
  
 *self*.\_\_semaphore.release()  
  
 *def* \_\_rpc\_caller(*self*) -> *None*:  
 *"""  
 This is ran on each thread in parallel inside the Router; it is an infinite loop that  
 exists for as long as the Router is running. It will check for work from the semaphore  
 – if there is no work it will wait until there is. Once there is work it will dequeue  
 an item from the queue and get K nodes closer to “key” than “contact”, updating the  
 FindResult – this works even though it has been dequeued because python refers to  
 lists as references (for example, if you pass a list into a function, the function  
 can edit the list, and that will persist outside the function), so we can refer to  
 item[“findResult”] because the reference will persist in the “lookup” method.  
 :return:  
 """* flag = *True  
 while* flag: *# I hate this.  
 self*.\_\_semaphore.acquire()  
 item: ContactQueueItem = *self*.\_\_contact\_queue.dequeue()  
 *if* item:  
 found, val, found\_by, item["closer\_contacts"], item["further\_contacts"] = *self*.get\_closer\_nodes(  
 item["key"],  
 item["contact"],  
 item["rpc\_call"],  
 item["closer\_contacts"],  
 item["further\_contacts"]  
 )  
 *if* val *or* found\_by:  
 *if not self*.\_\_stop\_work:  
 *# Possible multiple "found"  
 # lock(locker)* item["find\_result"]["found"] = *True* item["find\_result"]["found\_by"] = found\_by  
 item["find\_result"]["val"] = val  
 item["find\_result"]["contacts"] = item["closer\_contacts"]  
  
 *def* set\_query\_time(*self*) -> *None*:  
 *"""  
 Sets self.now() to current time.  
 :return:  
 """  
 self*.\_\_now = datetime.now()  
  
 *def* \_query\_time\_expired(*self*) -> bool:  
 *"""  
 Returns true if the query time has expired.  
  
 Returns if the time since query was triggered is longer than Constants REQUEST-TIMEOUT.  
 :return:  
 """  
 return* (datetime.now() - *self*.\_\_now).total\_seconds() > Constants.REQUEST\_TIMEOUT  
  
 *def* \_\_dequeue\_remaining\_work(*self*):  
 *"""  
 Dequeues everything from the contact queue.  
 :return:  
 """* dequeue\_result = *True  
 while* dequeue\_result:  
 dequeue\_result = *self*.\_\_contact\_queue.dequeue()  
  
 *def* \_stop\_remaining\_work(*self*):  
 *"""  
 Dequeues everything from the contact queue, then sets “stop work” to True, so no more work will be done.  
 :return:  
 """  
 self*.\_\_dequeue\_remaining\_work()  
 *self*.\_\_stop\_work = *True* @classmethod  
 *def* parallel\_found(*cls*, find\_result: FindResult, found\_ret: FindResult) -> tuple[bool, FindResult]:  
 *"""  
 Overlays found-ret with find-result if find-result has a value.  
 It then returns if the overlay has been performed, and the new found\_ret.  
 :param find\_result:  
 :param found\_ret:  
 :return:  
 """  
 # lock(locker)  
 if* find\_result["found"]:  
 *# lock(find\_result["contacts"]  
 # lock found ret* found\_ret["found"] = *True* found\_ret["contacts"] = find\_result["contacts"]  
 found\_ret["found\_by"] = find\_result["found\_by"]  
 found\_ret["val"] = find\_result["val"]  
  
 *return* find\_result["found"], found\_ret  
  
 *def* lookup(*self*, key: ID, rpc\_call: Callable, give\_me\_all: bool = *False*) -> FindResult:  
 *"""  
 Performs a similar algorithm to what has been performed in "Router", but this  
 time it is performed in parallel. First it makes sure that self.node actually exists  
 to prevent any strange errors from happening down the line. It then gets a list  
 of K 'all nodes' it intends to query, and then bits of chunks of ALPHA each time and  
 queries them individually. It then groups these ALPHA nodes\_to\_query into closer\_contacts  
 and further\_contacts, depending on if they are closer than or further than our ID to the  
 parameter "key" by the XOR metric. All the closer and further contacts are then placed  
 into a ContactQueueItem (and appended to our contact queue) with each member of the nodes  
 to query - This will be handled by the Constants.MAX\_THREADS running rpc\_caller. The  
 time since last query is then updated. This process then iterates, biting of chunks of  
 ALPHA contacts until there are no closer or further uncontacted nodes, or until one of the threads  
 handling the contact queue finds the value we are looking for which matches the key-value  
 pair with "key", if that is the case, the FindResult object containing the value will be returned.  
  
  
 :param key:  
 :param rpc\_call:  
 :param give\_me\_all:  
 :return:  
 """  
  
 if not* isinstance(*self*.node, Node):  
 *raise* TypeError("ParallelRouter must have instance node.")  
 have\_work: bool = *True* find\_result: FindResult = FindResult(found=*False*, found\_by=*None*, val="", contacts=[])  
 ret: list[Contact] = []  
 contacted\_nodes: list[Contact] = []  
 closer\_contacts: list[Contact] = []  
 further\_contacts: list[Contact] = []  
 found\_return = FindResult(found=*False*, found\_by=*None*, val="", contacts=[])  
  
 *if* Constants.DEBUG:  
 all\_nodes: list[Contact] = *self*.node.bucket\_list.get\_kbucket(key).contacts[0:Constants.K]  
 *else*:  
 *# For unit testing, this is a bad way to get a list of close contacts with virtual nodes  
 # because we're always going to get the closest nodes right at the get go.* all\_nodes: list[Contact] = *self*.node.bucket\_list.get\_close\_contacts(key, *self*.node.our\_contact.id)[0:Constants.K]  
  
 nodes\_to\_query: list[Contact] = all\_nodes[0:Constants.A]  
 *# Also not explicitly in specification:  
 # any closer node in the alpha list is immediately added to our closer contact list,  
 # and any further node in the alpha list is immediately added to our further contact list.  
 for* c *in* nodes\_to\_query:  
 *if* (c.id ^ key) < (*self*.node.our\_contact.id ^ key):  
 closer\_contacts.append(c)  
 *else*:  
 further\_contacts.append(c)  
 *# the remaining contacts can be put here.  
 for* c *in* all\_nodes:  
 *if* c *not in* nodes\_to\_query:  
 further\_contacts.append(c)  
 *# we're about to contact these nodes.  
 for* c *in* nodes\_to\_query:  
 *if* c.id *not in* [i.id *for* i *in* contacted\_nodes]:  
 contacted\_nodes.append(c)  
  
 *# Spec: the initiator then sends parallel asynchronous FIND\_NODE RPCs to the  
 # Constants.A nodes it has chosen.  
 for* c *in* nodes\_to\_query:  
 *self*.queue\_work(key=key,  
 contact=c,  
 rpc\_call=rpc\_call,  
 closer\_contacts=closer\_contacts,  
 further\_contacts=further\_contacts,  
 find\_result=find\_result)  
  
 *self*.set\_query\_time()  
 *# add any new closer contacts to the list we're going to return.  
 for* c *in* closer\_contacts:  
 *if* c.id *not in* [r.id *for* r *in* ret]:  
 ret.append(c)  
  
 *# The lookup terminates when the initiator has queried and  
 # received responses from the k closest nodes it has seen.  
 while* len(ret) < Constants.K *and* have\_work:  
 sleep(Constants.RESPONSE\_WAIT\_TIME / 1000)  
  
 found, found\_return = *self*.parallel\_found(find\_result, found\_return)  
 *if* found:  
 *self*.\_stop\_remaining\_work()  
 *return* found\_return  
  
 closer\_uncontacted\_nodes = [c *for* c *in* closer\_contacts *if* c *not in* contacted\_nodes]  
 further\_uncontacted\_nodes = [c *for* c *in* further\_contacts *if* c *not in* contacted\_nodes]  
  
 have\_closer = len(closer\_uncontacted\_nodes) > 0  
 have\_further = len(further\_uncontacted\_nodes) > 0  
 have\_work = have\_closer *or* have\_further *or not self*.\_query\_time\_expired()  
  
 *# for the k nodes the initiator has heard of closest to the target...* alpha\_nodes = *None  
  
 if* have\_closer:  
 *# we're about to contact these nodes.  
 if* len(closer\_uncontacted\_nodes) >= Constants.A:  
 alpha\_nodes = closer\_uncontacted\_nodes[0: Constants.A - 1]  
 *else*:  
 alpha\_nodes = closer\_uncontacted\_nodes  
  
 *if* alpha\_nodes:  
 *for* a *in* alpha\_nodes:  
 *if* a.id *not in* [c.id *for* c *in* contacted\_nodes]:  
 contacted\_nodes.append(a)  
 *self*.queue\_work(  
 key=key,  
 contact=a,  
 rpc\_call=rpc\_call,  
 closer\_contacts=closer\_contacts,  
 further\_contacts=further\_contacts,  
 find\_result=find\_result  
 )  
 *self*.set\_query\_time()  
  
 *elif* have\_further:  
 *if* len(further\_uncontacted\_nodes) >= Constants.A:  
 alpha\_nodes = further\_uncontacted\_nodes[0: Constants.A - 1]  
 *else*:  
 alpha\_nodes = further\_uncontacted\_nodes  
  
 *if* alpha\_nodes:  
 *for* a *in* alpha\_nodes:  
 *if* a.id *not in* [c.id *for* c *in* contacted\_nodes]:  
 contacted\_nodes.append(a)  
 *self*.queue\_work(  
 key=key,  
 contact=a,  
 rpc\_call=rpc\_call,  
 closer\_contacts=closer\_contacts,  
 further\_contacts=further\_contacts,  
 find\_result=find\_result  
 )  
 *self*.set\_query\_time()  
  
  
 *self*.\_stop\_remaining\_work()  
 *return* FindResult(  
 found=*False*,  
 contacts=ret *if* give\_me\_all *else* sorted(ret[0:Constants.K], key=*lambda* c: c.id ^ key),  
 found\_by=*None*,  
 val=*None* )

### storage.py

*import* os  
*from* datetime *import* datetime  
*import* json  
*from* typing *import* Optional  
  
*from* kademlia *import* pickler  
*from* kademlia.constants *import* Constants  
*from* kademlia.dictionaries *import* StoreValue  
*from* kademlia.id *import* ID  
*from* kademlia.interfaces *import* IStorage  
  
  
*class* VirtualStorage(IStorage):  
 *"""  
 Simple storage mechanism that stores things in memory.  
 """  
  
 def* \_\_init\_\_(*self*):  
 *self*.\_store: dict[int, StoreValue] = {}  
  
 *def* contains(*self*, key: ID) -> bool:  
 *"""  
 Returns a Boolean stating whether a key-value pair exists, given key.  
 """  
 return* key.value *in self*.get\_keys()  
  
 *def* get(*self*, key: ID | int) -> str:  
 *"""  
 Returns stored value, associated with given key value.  
 :param key: Type ID or Integer, key value to be searched.  
 :return:  
 """  
 if* isinstance(key, ID):  
 *return self*.\_store[key.value]["value"]  
 *elif* isinstance(key, int):  
 *return self*.\_store[key]["value"]  
 *else*:  
 *raise* TypeError("'get()' parameter 'key' must be type ID or int.")  
  
 *def* get\_timestamp(*self*, key: int) -> datetime:  
 *"""  
 Returns when the key was last republished as a datetime object.  
 :param key:  
 :return:  
 """  
 return* datetime.fromisoformat(*self*.\_store[key]["republish\_timestamp"])  
  
 *def* set(*self*, key: ID, value: str, expiration\_time\_sec: int = 0) -> *None*:  
 *"""  
 Stores a key value pair, along with the expiration time and timestamp.  
 :param key:  
 :param value:  
 :param expiration\_time\_sec:  
 :return:  
 """  
 self*.\_store[key.value] = StoreValue(value=value,  
 expiration\_time=expiration\_time\_sec,  
 republish\_timestamp=datetime.now().isoformat()  
 )  
 *self*.touch(key.value)  
  
 *def* get\_expiration\_time\_sec(*self*, key: int) -> int:  
 *"""  
 Returns how long it takes for the given key-value pair to expire, given key.  
 :param key:  
 :return:  
 """  
 return self*.\_store[key]["expiration\_time"]  
  
 *def* remove(*self*, key: int) -> *None*:  
 *"""  
 Removes a given key-value pair, given key.  
 :param key:  
 :return:  
 """  
 if* key *in self*.\_store:  
 *self*.\_store.pop(key, *None*)  
  
 *def* get\_keys(*self*) -> list[int]:  
 *"""  
 Returns all keys of key-value pairs that are stored.  
 :return:  
 """  
 return* list(*self*.\_store.keys())  
  
 *def* touch(*self*, key: int) -> *None*:  
 *"""  
 “touches” a given key-value pair, this is done by updating the timestamp to the current time.  
 :param key:  
 :return:  
 """  
 self*.\_store[key]["republish\_timestamp"] = datetime.now().isoformat()  
  
 *def* try\_get\_value(*self*, key: ID) -> tuple[bool, str | *None*]:  
 *"""  
 Tries to get a given value from a key-value pair, given the key. Returns True | False,  
 and the value if it was found.  
 :param key:  
 :return:  
 """* val: Optional[str] = *None* ret = *False  
 if* key.value *in self*.\_store:  
 val = *self*.\_store[key.value]["value"]  
 ret = *True  
  
 return* ret, val  
  
  
*class* SecondaryJSONStorage(IStorage):  
 *def* \_\_init\_\_(*self*, filename: str):  
 *"""  
 Storage object which reads/writes to a JSON file instead of to memory like how VirtualStorage does.  
 the JSON is formatted as dict[int, StoreValue].  
  
 :param filename: Filename to save values to - must end in .json!  
 """  
 self*.filename = filename  
 *if not* os.path.exists(*self*.filename):  
 cwd = os.getcwd()  
 *if not* os.path.exists(os.path.join(cwd, os.path.dirname(*self*.filename))):  
 os.mkdir(os.path.join(cwd, os.path.dirname(*self*.filename)))  
 *with* open(*self*.filename, "w"):  
 *pass # Makes file.  
  
 def* set(*self*, key: ID, value: str | bytes, expiration\_time\_sec: int = 0) -> *None*:  
 *"""  
 Sets a key-value pair in the JSON along with the expiration time in seconds,  
 and the timestamp as the current time. The python JSON library cannot store  
 datetime objects, so it is converted in and out of “ISOFormat” which is a  
 string representation of it.  
 :param key:  
 :param value:  
 :param expiration\_time\_sec:  
 :return:  
 """* os.makedirs(os.path.dirname(*self*.filename), exist\_ok=*True*)  
 *with* open(*self*.filename, "r") *as* f:  
 print(f"Set at {*self*.filename}.")  
 *try*:  
 json\_data: dict = json.load(f)  
 *except* json.JSONDecodeError:  
 json\_data = {}  
  
 to\_store: StoreValue = StoreValue(  
 value=value,  
 expiration\_time=expiration\_time\_sec,  
 republish\_timestamp=datetime.now().isoformat()  
 )  
 *if* key.value *in* json\_data:  
 json\_data.pop(key.value)  
 *if* str(key.value) *in* json\_data:  
 json\_data.pop(str(key.value))  
  
 print(json\_data)  
 json\_data[key.value] = to\_store  
 print(json\_data)  
  
 *with* open(*self*.filename, "w") *as* f:  
 json.dump(json\_data, f)  
  
 *def* contains(*self*, key: ID | int) -> bool:  
 *"""  
 Returns if the storage file contains a key-value pair, given the key.  
 :param key:  
 :return:  
 """  
 with* open(*self*.filename, "r") *as* f:  
 print(f"Contains at {*self*.filename}")  
 f.seek(0)  
 *try*:  
 json\_data: dict[int, StoreValue] = json.load(f)  
 *except* json.JSONDecodeError *as* e:  
 print(e)  
 json\_data = {}  
  
 *if* isinstance(key, ID):  
 *return* str(key.value) *in* list(json\_data.keys())  
 *else*:  
 *return* str(key) *in* list(json\_data.keys())  
  
 *def* get\_timestamp(*self*, key: int | ID) -> datetime:  
 *"""  
 Gets the timestamp of a key-value pair, given the key.  
 :param key:  
 :return:  
 """  
 with* open(*self*.filename, "r") *as* f:  
 print(f"Get timestamp at {*self*.filename}")  
 *try*:  
 json\_data: dict[int, StoreValue] = json.load(f)  
 *except* json.JSONDecodeError *as* e:  
 print(e)  
 json\_data = {}  
 *if* isinstance(key, ID):  
 *return* datetime.fromisoformat(json\_data[key.value]["republish\_timestamp"])  
 *else*:  
 *return* datetime.fromisoformat(json\_data[key]["republish\_timestamp"])  
  
 *def* get(*self*, key: ID | int) -> str:  
 *"""  
 Returns the value of a key-value pair from the storage file, given the key.  
 :param key:  
 :return:  
 """  
 with* open(*self*.filename, "r") *as* f:  
 f.seek(0)  
 print(f"Get at {*self*.filename}")  
 *# try:* json\_data: dict = json.load(f)  
 print("fdata", json\_data)  
 *# except json.JSONDecodeError as e:  
 # print(e)  
 # json\_data = {}  
 if* isinstance(key, ID):  
 *return* json\_data[str(key.value)]["value"]  
 *elif* isinstance(key, int):  
 *return* json\_data[str(key)]["value"]  
 *else*:  
 *raise* TypeError("'get()' parameter 'key' must be type ID or int.")  
  
 *def* get\_expiration\_time\_sec(*self*, key: int) -> int:  
 *"""  
 Gets the time to expire for a key-value pair, given the key.  
 :param key:  
 :return:  
 """  
 with* open(*self*.filename, "r") *as* f:  
 print(f"Get expiration time at {*self*.filename}")  
 *try*:  
 json\_data: dict[int, StoreValue] = json.load(f)  
 *except* json.JSONDecodeError:  
 json\_data = {}  
 *return* json\_data[key]["expiration\_time"]  
  
 *def* remove(*self*, key: int) -> *None*:  
 *"""  
 Removes a key-value pair, given the key.  
 :param key:  
 :return:  
 """  
 with* open(*self*.filename, "r") *as* f:  
 print(f"Remove at {*self*.filename}")  
 *try*:  
 json\_data: dict[str, StoreValue] = json.load(f)  
 *except* json.JSONDecodeError:  
 json\_data = {}  
  
 *if* str(key) *in* json\_data:  
 json\_data.pop(str(key), *None*)  
  
 *with* open(*self*.filename, "w") *as* f:  
 json.dump(json\_data, f)  
  
 *def* get\_keys(*self*) -> list[int]:  
 *"""  
 Returns all keys stored by the storage file as a list of integers.  
 :return:  
 """  
 with* open(*self*.filename, "r") *as* f:  
 print(f"Get keys at {*self*.filename}")  
 *try*:  
 json\_data: dict[int, StoreValue] = json.load(f)  
 *except* json.JSONDecodeError:  
 json\_data = {}  
 *return* list(json\_data.keys())  
  
 *def* touch(*self*, key: int | ID) -> *None*:  
 *"""  
 “touches” a key-value pair by setting the timestamp to the current time.  
 :param key:  
 :return:  
 """  
 with* open(*self*.filename, "r") *as* f:  
 print(f"Touch at {*self*.filename}")  
 *try*:  
 json\_data: dict[int, StoreValue] = json.load(f)  
 *except* json.JSONDecodeError:  
 json\_data = {}  
 *if* isinstance(key, ID):  
 json\_data[key.value]["republish\_timestamp"] = datetime.now().isoformat()  
 *else*:  
 json\_data[key]["republish\_timestamp"] = datetime.now().isoformat()  
 *with* open(*self*.filename, "w") *as* f:  
 json.dump(json\_data, f)  
  
 *def* try\_get\_value(*self*, key: ID) -> tuple[bool, int | str]:  
  
 *with* open(*self*.filename, "r") *as* f:  
 print(f"Try get value at {*self*.filename}")  
 *try*:  
 f.seek(0)  
 print("File:", f.read())  
 f.seek(0)  
 *# Key is a string because JSON library stores integers at strings* json\_data: dict[str, StoreValue] = json.load(f)  
 print(json\_data)  
 *except* json.JSONDecodeError *as* e:  
 print(e)  
 json\_data = {}  
 val = *None* ret = *False  
 if* str(key.value) *in* json\_data:  
 val = json\_data[str(key.value)]["value"]  
 ret = *True  
 if* json\_data != {}:  
 *with* open(*self*.filename, "w") *as* f:  
 json.dump(json\_data, f)  
  
 *return* ret, val  
  
 *def* set\_file(*self*, key: ID, filename: str, expiration\_time\_sec: int = 0) -> *None*:  
 *"""  
 Adds a file to storage file, it does this by loading ALL of the file to be added to memory,  
 and then pasting it into the storage file ALSO loaded into memory D:  
 :param key:  
 :param filename:  
 :param expiration\_time\_sec:  
 :return:  
 """  
 with* open(filename) *as* f:  
 print(f"Set file at {*self*.filename}")  
 file\_data = f.read()  
 data\_dict = {"filename": filename, "file\_data": file\_data}  
 encoded\_data: bytes = pickler.plain\_encode\_data(data=data\_dict)  
 encoded\_data\_str = encoded\_data.decode(Constants.PICKLE\_ENCODING)  
 *self*.set(  
 key=key,  
 value=encoded\_data\_str,  
 expiration\_time\_sec=expiration\_time\_sec  
 )

### GUI.py

*import* os  
*import* pickle  
*import* re  
*import* threading  
*import* json  
*from* os.path *import* exists, isfile  
*import* argparse  
  
*import* customtkinter *as* ctk  
*from* PIL *import* Image  
*from* requests *import* get  
  
*from* kademlia *import* dht, id, networking, protocols, node, contact, storage, routers, errors, helpers  
*from* kademlia.constants *import* Constants  
  
"""  
├── User Interface  
│ ├── Join  
│ │ ├── Settings  
│ │ ├── Load an existing network  
│ │ └── Bootstrap into a new network  
│ ├── Main menu  
│ │ ├── Settings  
│ │ ├── Download a file  
│ │ ├── Add a file for upload  
│ │ ├── Remove a file for upload  
│ │ ├── Leave network  
│ │ └── Search using key  
  
  
open UserInterface(), then check if the user is in a network already or not  
- the k-buckets are stored in a JSON file, which is used to check this,   
and if the user is not in a network, the user is prompted to join a network.  
"""  
parser = argparse.ArgumentParser()  
parser.add\_argument("--use\_global\_ip", action="store\_true",  
 help="If the clients global IP should be used by the P2P network.")  
parser.add\_argument("--port", type=int, required=*False*, default=7124)  
  
args = parser.parse\_args()  
  
USE\_GLOBAL\_IP = args.use\_global\_ip  
PORT = args.port  
  
  
*class* Fonts:  
 title\_font = ("Segoe UI", 20, "bold")  
 text\_font = ("Segoe UI", 16)  
  
  
*class* ContactViewer(ctk.CTk):  
 *def* \_\_init\_\_(*self*, id: int, protocol\_type: type, url: str, port: int, appearance\_mode="dark"):  
 super().\_\_init\_\_()  
 ctk.set\_appearance\_mode(appearance\_mode)  
  
 *self*.id = id  
 *self*.url = url  
 *self*.port = port  
 *self*.protocol\_type = protocol\_type  
  
 *self*.contact\_viewer\_title = ctk.CTkLabel(*self*, text="Our Contact:", font=Fonts.title\_font)  
 *self*.contact\_viewer\_title.pack(padx=20, pady=30)  
  
 *self*.id\_label = ctk.CTkLabel(*self*, text=f"ID: {*self*.id}", font=Fonts.text\_font)  
 *self*.id\_label.pack(padx=20, pady=10)  
  
 *self*.protocol\_type\_label = ctk.CTkLabel(*self*, text=f"Protocol type: {*self*.protocol\_type}", font=Fonts.text\_font)  
 *self*.protocol\_type\_label.pack(padx=20, pady=10)  
  
 *self*.url\_label = ctk.CTkLabel(*self*, text=f"URL: {*self*.url}", font=Fonts.text\_font)  
 *self*.url\_label.pack(padx=20, pady=10)  
  
 *self*.port\_label = ctk.CTkLabel(*self*, text=f"Port: {*self*.port}", font=Fonts.text\_font)  
 *self*.port\_label.pack(padx=20, pady=10)  
  
 *self*.export\_button = ctk.CTkButton(*self*, text="Export our contact", font=Fonts.text\_font,  
 command=*self*.export\_contact)  
 *self*.export\_button.pack(padx=20, pady=10)  
  
 *def* show\_error(*self*, error\_message: str):  
 print(f"[Error] {error\_message}")  
 error\_window = ErrorWindow(error\_message)  
 error\_window.mainloop()  
  
 *def* export\_contact(*self*, filename="our\_contact.json"):  
 contact\_dict = {  
 "url": *self*.url,  
 "port": *self*.port,  
 "protocol\_type": str(*self*.protocol\_type),  
 "id": *self*.id  
 }  
 print("[Status] Exporting our contact...")  
 *with* open(filename, "w") *as* f:  
 json.dump(contact\_dict, f)  
 *self*.show\_status(f"Exported our contact to {filename}.")  
  
 *def* show\_status(*self*, message: str):  
 print(f"[Status] {message}")  
 status\_window = StatusWindow(message)  
 status\_window.mainloop()  
  
  
*class* StatusWindow(ctk.CTk):  
 *def* \_\_init\_\_(*self*, message: str, copy\_data=*None*):  
 *"""  
 Creates the status window, with option for copying data to clipboard if there is copy data.  
 :param message:  
 :param copy\_data:  
 """* ctk.CTk.\_\_init\_\_(*self*)  
 *self*.copy\_data = copy\_data  
 *self*.message = ctk.CTkLabel(*self*, text=message, font=Fonts.text\_font)  
 *self*.message.pack(padx=30, pady=20)  
 *self*.copy\_button = ctk.CTkButton(*self*, text="Copy to clipboard", font=Fonts.text\_font,  
 command=*self*.copy)  
 *if* copy\_data:  
 *self*.copy\_button.pack(padx=30, pady=20)  
  
 *def* copy(*self*):  
 print(f"[GUI] Copying data to clipboard: {*self*.copy\_data}")  
 *self*.clipboard\_clear()  
 *self*.clipboard\_append(*self*.copy\_data)  
 *self*.update()  
  
  
*class* Settings(ctk.CTk):  
 *def* \_\_init\_\_(*self*, hash\_table: dht.DHT | *None*, appearance\_mode="dark"):  
 super().\_\_init\_\_()  
 ctk.set\_appearance\_mode(appearance\_mode)  
  
 *self*.appearance\_mode = appearance\_mode  
  
 *self*.dht: dht.DHT | *None* = hash\_table  
  
 *self*.title("Kademlia Settings")  
  
 *self*.settings\_title = ctk.CTkLabel(*self*, text="Settings", font=Fonts.title\_font)  
 *self*.settings\_title.grid(column=0, row=0, columnspan=2, padx=20, pady=20)  
  
 *if self*.dht:  
 *self*.dht\_export\_label = ctk.CTkLabel(*self*, text="File to export to:", width=150, font=Fonts.text\_font)  
 *self*.dht\_export\_label.grid(column=0, row=1, padx=10, pady=10)  
  
 *self*.dht\_export\_file = ctk.CTkEntry(*self*, width=200, height=20, font=Fonts.text\_font)  
 *self*.dht\_export\_file.grid(column=1, row=1, padx=10, pady=10)  
 *self*.dht\_export\_file.insert("1", f"dht.pickle")  
 print("CTK BUTTON TYPE", type(ctk.CTkButton))  
 *self*.export\_dht\_button = ctk.CTkButton(*self*, text="Export/Save DHT", font=Fonts.text\_font,  
 command=*self*.export\_dht)  
 *self*.export\_dht\_button.grid(column=1, row=2, padx=10, pady=10)  
  
 *self*.view\_contact\_button = ctk.CTkButton(*self*, text="View our contact", font=Fonts.text\_font,  
 command=*self*.view\_contact)  
 *self*.view\_contact\_button.grid(column=0, row=2, padx=10, pady=10)  
 *else*:  
 no\_dht\_label = ctk.CTkLabel(*self*,  
 text="You have not made a DHT yet! You should not be able to access this.")  
 no\_dht\_label.grid(column=0, row=1, padx=10, pady=10)  
  
 *def* show\_error(*self*, error\_message: str):  
 print(f"[Error] {error\_message}")  
 error\_window = ErrorWindow(error\_message)  
 error\_window.mainloop()  
  
 *def* show\_status(*self*, message: str):  
 print(f"[Status] {message}")  
 status\_window = StatusWindow(message)  
 status\_window.mainloop()  
  
 *def* export\_dht(*self*):  
 *try*:  
 file = *self*.dht\_export\_file.get().strip("\n")  
 *self*.dht.save(file)  
 *self*.show\_status(f"File saved successfully to {file}.")  
 *except* Exception *as* e:  
 *self*.show\_error(str(e))  
  
 *def* view\_contact(*self*):  
 our\_contact: contact.Contact = *self*.dht.our\_contact  
 our\_id: int = our\_contact.id.value  
 *# noinspection PyTypeChecker* protocol: protocols.TCPProtocol = our\_contact.protocol  
 protocol\_type: type = type(protocol)  
 our\_ip\_address: str = protocol.url  
 our\_port: int = protocol.port  
  
 contact\_viewer = ContactViewer(  
 id=our\_id,  
 protocol\_type=protocol\_type,  
 url=our\_ip\_address,  
 port=our\_port,  
 appearance\_mode=*self*.appearance\_mode  
 )  
 contact\_viewer.mainloop()  
  
  
*class* ErrorWindow(ctk.CTk):  
 *def* \_\_init\_\_(*self*, error\_message: str):  
 super().\_\_init\_\_()  
 *self*.title = ctk.CTkLabel(*self*, text="Error", font=Fonts.title\_font)  
 *self*.title.pack(padx=20, pady=20)  
  
 *self*.error\_message = ctk.CTkLabel(*self*, text=error\_message, font=Fonts.text\_font)  
 *self*.error\_message.pack(padx=20, pady=10)  
  
  
*class* MainGUI(ctk.CTk):  
 *def* \_\_init\_\_(*self*, appearance\_mode="dark"):  
 ctk.CTk.\_\_init\_\_(*self*)  
 *self*.settings\_button = *None  
 self*.appearance\_mode = appearance\_mode  
 ctk.set\_appearance\_mode(appearance\_mode)  
 *# self.geometry("600x500")  
 self*.title("Kademlia")  
  
 *# Create our contact - this should be overwritten if bootstrapping.  
 # self.initialise\_kademlia()  
  
 self*.make\_join\_dht\_frame()  
  
 *def* initialise\_kademlia(*self*):  
 *"""  
 Creates DHT, server and server thread. If GET-GLOBAL-IP is true, then it will get our global IP by  
 decoding a response from ‘https://api.ipify.org’, which according to a StackOverflow article is the  
 most efficient way to get your global IP in python. If GET-GLOBAL-IP is false, IP is set to “127.0.0.1”.  
 This is useful for DHTs on just the local networks, so no port forwarding needs to be set up.  
  
 Then a valid port is attempted to be found by getting a random integer between 5000 and 35000,  
 until the port is free. This allows for multiple instances on one device, because the port is not  
 hard coded. A TCPProtocol is created with this IP and Protocol. Then a contact is created with a  
 random ID, and the protocol we just created. A JSON storage object is setup for main storage, and  
 VirtualStorage for cache storage. These are placed into a subfolder with title “id.value”.  
  
 Now we have initialised Kademlia, the main network frame is launched.  
  
 :return:  
 """* print("[Initialisation] Initialising Kademlia.")  
  
 our\_id = id.ID.random\_id()  
 *if* USE\_GLOBAL\_IP: *# Port forwarding is required.* our\_ip = get('https://api.ipify.org').content.decode('utf8')  
 *else*:  
 our\_ip = "127.0.0.1"  
 print(f"[Initialisation] Our hostname is {our\_ip}.")  
  
 *if* PORT:  
 valid\_port = PORT  
 *else*:  
 valid\_port = helpers.get\_valid\_port()  
  
 print(f"[Initialisation] Port free at {valid\_port}, creating our node here.")  
  
 protocol = protocols.TCPProtocol(  
 url=our\_ip, port=valid\_port  
 )  
  
 our\_node = node.Node(  
 contact=contact.Contact(  
 id=our\_id,  
 protocol=protocol  
 ),  
 storage=storage.SecondaryJSONStorage(f"{our\_id.value}/node.json"),  
 cache\_storage=storage.VirtualStorage()  
 )  
  
 *# Make directory of our\_id at current working directory.* create\_dir\_at = os.path.join(os.getcwd(), str(our\_id.value))  
 print("[GUI] Making directory at", create\_dir\_at)  
 *if not* exists(create\_dir\_at):  
 os.mkdir(create\_dir\_at)  
 *self*.dht: dht.DHT = dht.DHT(  
 id=our\_id,  
 protocol=protocol,  
 originator\_storage=storage.SecondaryJSONStorage(f"{our\_id.value}/originator\_storage.json"),  
 republish\_storage=storage.SecondaryJSONStorage(f"{our\_id.value}/republish\_storage.json"),  
 cache\_storage=storage.VirtualStorage(),  
 router=routers.ParallelRouter(our\_node)  
 )  
  
 *self*.server = networking.TCPServer(our\_node)  
 *self*.server\_thread = *self*.server.thread\_start()  
  
 *self*.make\_network\_frame()  
  
 *def* open\_settings(*self*):  
 *if* hasattr(*self*, "dht"):  
 settings\_window = Settings(hash\_table=*self*.dht, appearance\_mode=*self*.appearance\_mode)  
 settings\_window.mainloop()  
 *else*:  
 *pass  
  
 def* thread\_open\_settings(*self*):  
 *"""  
 OBSELETE - open\_settings works fine.  
 Opens the server window in a thread - this is not recommended to use.  
 :return:  
 """* settings\_thread = threading.Thread(target=*self*.open\_settings, daemon=*True*)  
 settings\_thread.daemon = *True # Dies when program ends.* settings\_thread.start()  
  
 *def* add\_settings\_icon(*self*):  
 dark\_icon = Image.open(r"assets/settings\_icon\_light.png")  
 light\_icon = Image.open(r"assets/settings\_icon\_dark.png")  
 settings\_icon = ctk.CTkImage(light\_image=light\_icon, dark\_image=dark\_icon, size=(30, 30))  
 *self*.settings\_button = ctk.CTkButton(*self*, image=settings\_icon, text="",  
 bg\_color="transparent", fg\_color="transparent",  
 width=28, command=*self*.open\_settings)  
  
 *self*.settings\_button.pack(side=ctk.BOTTOM, anchor=ctk.S, padx=10, pady=10)  
  
 *def* clear\_screen(*self*):  
 *for* child *in self*.winfo\_children():  
 child.destroy()  
  
 *def* clear\_screen\_and\_keep\_settings(*self*):  
 *self*.clear\_screen()  
 *if* hasattr(*self*, "dht"):  
 *self*.add\_settings\_icon()  
  
 *def* make\_join\_dht\_frame(*self*):  
 *self*.clear\_screen\_and\_keep\_settings()  
 join = JoinNetworkMenuFrame(parent=*self*)  
 join.pack(padx=20, pady=20)  
  
 *def* make\_load\_dht\_frame(*self*):  
 *self*.clear\_screen\_and\_keep\_settings()  
 load\_dht = LoadDHTFromFileFrame(parent=*self*)  
 load\_dht.pack(padx=20, pady=20)  
  
 *def* make\_bootstrap\_frame(*self*):  
 *self*.clear\_screen\_and\_keep\_settings()  
 bootstrap = BootstrapFrame(parent=*self*)  
 bootstrap.pack(padx=20, pady=20)  
  
 *def* make\_bootstrap\_from\_json\_frame(*self*):  
 *self*.clear\_screen\_and\_keep\_settings()  
 bootstrap\_from\_json = BootstrapFromJSONFrame(parent=*self*)  
 bootstrap\_from\_json.pack(padx=20, pady=20)  
  
 *def* make\_network\_frame(*self*):  
 *"""  
 Main network page  
 I want this to have the following buttons:  
 - Download file  
 - Add new file  
 :return:  
 """  
 self*.clear\_screen\_and\_keep\_settings()  
 network\_frame = MainNetworkFrame(*self*)  
 network\_frame.pack(padx=20, pady=20)  
  
 @classmethod  
 *def* show\_error(*cls*, error\_message: str):  
 print(f"[Error] {error\_message}")  
 error\_window = ErrorWindow(error\_message)  
 error\_window.mainloop()  
  
 @classmethod  
 *def* show\_status(*cls*, message: str, copy\_data=*None*):  
 print(f"[Status] {message}")  
 status\_window = StatusWindow(message, copy\_data)  
 status\_window.mainloop()  
  
 *def* make\_download\_frame(*self*):  
 *self*.clear\_screen\_and\_keep\_settings()  
 download\_frame = DownloadFrame(*self*)  
 download\_frame.pack(padx=20, pady=20)  
  
 *def* make\_upload\_frame(*self*):  
 *self*.clear\_screen\_and\_keep\_settings()  
 upload\_frame = UploadFrame(*self*)  
 upload\_frame.pack(padx=20, pady=20)  
  
  
*class* UploadFrame(ctk.CTkFrame):  
 *def* \_\_init\_\_(self, parent: MainGUI, fg\_color="transparent", \*\*kwargs):  
 ctk.CTkFrame.\_\_init\_\_(self, parent, \*\*kwargs)  
 self.configure(fg\_color=fg\_color)  
 self.parent = parent  
  
 self.title = ctk.CTkLabel(self, text="Upload File", font=Fonts.title\_font)  
 self.title.grid(column=0, row=0, columnspan=2, padx=20, pady=10)  
  
 self.enter\_file\_label = ctk.CTkLabel(self, text="File to upload:", font=Fonts.text\_font)  
 self.enter\_file\_label.grid(column=0, row=1, padx=20, pady=10)  
  
 self.enter\_file\_entry = ctk.CTkEntry(self, width=150, height=20, font=Fonts.text\_font)  
 self.enter\_file\_entry.grid(column=1, row=1, padx=20, pady=10)  
  
 self.back\_button = ctk.CTkButton(self, text="Back", font=Fonts.text\_font,  
 command=self.parent.make\_network\_frame)  
 self.back\_button.grid(column=0, row=2, padx=20, pady=10)  
  
 self.upload\_button = ctk.CTkButton(self, text="Upload", font=Fonts.text\_font,  
 command=self.handle\_upload)  
 self.upload\_button.grid(column=1, row=2, columnspan=1, padx=20, pady=10)  
  
 *def* handle\_upload(self):  
 file\_to\_upload = self.enter\_file\_entry.get().strip("\n")  
 *if* isfile(file\_to\_upload):  
 filename = os.path.basename(file\_to\_upload)  
 *if not* filename: *# os.path.basename returns "" on file paths ending in "/"* self.parent.show\_error("Must not be a directory.")  
 *else*:  
 *with* open(file\_to\_upload, "rb") *as* f:  
 file\_contents: bytes = f.read()  
 *# val will be a 'latin1' pickled dictionary {filename: str, file: bytes}* val: str = pickle.dumps({"filename": filename, "file": file\_contents}).decode(Constants.PICKLE\_ENCODING)  
 id\_to\_store\_to = id.ID.random\_id()  
 self.parent.dht.store(id\_to\_store\_to, val)  
 self.parent.show\_status(f"Stored file at {id\_to\_store\_to}.", copy\_data=str(id\_to\_store\_to))  
 *else*:  
 self.parent.show\_error(f"Path not found: {file\_to\_upload}")  
  
  
*class* DownloadFrame(ctk.CTkFrame):  
 *def* \_\_init\_\_(self, parent: MainGUI, fg\_color="transparent", \*\*kwargs):  
 ctk.CTkFrame.\_\_init\_\_(self, parent, \*\*kwargs)  
 self.configure(fg\_color=fg\_color)  
 self.parent = parent  
  
 self.title = ctk.CTkLabel(self, text="Download File", font=Fonts.title\_font)  
 self.title.grid(column=0, row=0, columnspan=2, padx=20, pady=10)  
  
 self.enter\_id\_label = ctk.CTkLabel(self, text="ID to download:", font=Fonts.text\_font)  
 self.enter\_id\_label.grid(column=0, row=1, padx=20, pady=10)  
  
 self.enter\_id\_entry = ctk.CTkEntry(self, width=150, height=20, font=Fonts.text\_font)  
 self.enter\_id\_entry.grid(column=1, row=1, padx=20, pady=10)  
  
 self.back\_button = ctk.CTkButton(self, text="Back", font=Fonts.text\_font,  
 command=self.parent.make\_network\_frame)  
 self.back\_button.grid(column=0, row=2, padx=20, pady=10)  
  
 self.download\_button = ctk.CTkButton(self, text="Download", font=Fonts.text\_font,  
 command=self.handle\_download)  
 self.download\_button.grid(column=1, row=2, columnspan=1, padx=20, pady=10)  
  
 *def* handle\_download(self):  
 id\_from\_entry: str = self.enter\_id\_entry.get().strip("\n")  
  
 *if not* id\_from\_entry:  
 self.parent.show\_error("ID must not be empty.")  
 *elif not* id\_from\_entry.isnumeric():  
 self.parent.show\_error("ID was not a number.")  
 *elif not* 0 <= int(id\_from\_entry) < 2 \*\* Constants.ID\_LENGTH\_BITS:  
 self.parent.show\_error("ID out of range.")  
 *else*:  
 id\_to\_download: id.ID = id.ID(int(id\_from\_entry))  
 print("calling find value")  
 found, contacts, val = self.parent.dht.find\_value(key=id\_to\_download)  
 *# val will be a 'latin1' pickled dictionary {filename: str, file: bytes}  
 if not* found:  
 self.parent.show\_error("File not found.")  
 *else*:  
  
 val\_bytes: bytes = val.encode(Constants.PICKLE\_ENCODING)  
  
 *# "pickle.loads()" is very insecure and can lead to arbitrary code execution, the val received  
 # could be maliciously crafted to allow for malicious code execution because it compiles and creates  
 # a python object - I don't think this is an issue because it never calls any methods. If this does  
 # turn out to be dangerous, JSON should be used instead.* file\_dict: dict = pickle.loads(val\_bytes)  
 filename: str = file\_dict["filename"]  
 file\_bytes: bytes = file\_dict["file"]  
 *del* file\_dict *# Free up memory.* cwd = os.getcwd() *# An option could be added to change where it is installed to.  
 with* open(os.path.join(cwd, filename), "wb") *as* f:  
 f.write(file\_bytes)  
  
 self.parent.show\_status(f"File downloaded to {os.path.join(cwd, filename)}.")  
  
  
*class* MainNetworkFrame(ctk.CTkFrame):  
 *def* \_\_init\_\_(self, parent: MainGUI, fg\_color="transparent", \*\*kwargs):  
 ctk.CTkFrame.\_\_init\_\_(self, parent, \*\*kwargs)  
 self.configure(fg\_color=fg\_color)  
 self.parent = parent  
  
 self.title = ctk.CTkLabel(self, text="Kademlia", font=Fonts.title\_font)  
 self.title.grid(column=0, row=0, columnspan=2, padx=20, pady=10)  
  
 self.download\_button = ctk.CTkButton(self, text="Download", font=Fonts.text\_font,  
 command=self.parent.make\_download\_frame)  
 self.download\_button.grid(column=0, row=1, padx=10, pady=10)  
  
 self.upload\_button = ctk.CTkButton(self, text="Upload", font=Fonts.text\_font,  
 command=self.parent.make\_upload\_frame)  
 self.upload\_button.grid(column=1, row=1, padx=10, pady=10)  
  
  
*class* LoadDHTFromFileFrame(ctk.CTkFrame):  
 *def* \_\_init\_\_(self, parent: MainGUI, fg\_color="transparent", \*\*kwargs):  
 ctk.CTkFrame.\_\_init\_\_(self, parent, \*\*kwargs)  
 self.configure(fg\_color=fg\_color)  
 self.parent = parent  
  
 self.load\_title = ctk.CTkLabel(master=self, text="Load DHT from file", font=Fonts.title\_font)  
 self.load\_title.grid(column=0, row=0, columnspan=2, padx=20, pady=10)  
  
 self.enter\_file\_name\_text = ctk.CTkLabel(master=self, text="Load from file: ", font=Fonts.text\_font)  
 self.enter\_file\_name\_text.grid(column=0, row=1, padx=20, pady=20)  
  
 self.file\_name\_entry = ctk.CTkEntry(master=self, width=150, height=30, font=Fonts.text\_font)  
 self.file\_name\_entry.grid(column=1, row=1, padx=20, pady=20)  
  
 self.back\_button = ctk.CTkButton(master=self, text="Back", font=Fonts.text\_font,  
 command=self.parent.make\_join\_dht\_frame)  
 self.back\_button.grid(column=0, row=2, padx=20, pady=0)  
  
 self.load\_button = ctk.CTkButton(master=self, text="Load DHT", font=Fonts.text\_font,  
 command=self.load\_dht)  
 self.load\_button.grid(column=1, row=2, padx=20, pady=0)  
  
 *def* load\_dht(self):  
 filename = self.file\_name\_entry.get().strip("\n")  
  
 *if not* isfile(filename):  
 self.parent.show\_error(f"File not found:\n'{filename}'")  
 *return  
 try*:  
 loaded\_dht = dht.DHT.load(filename=filename)  
 *except* pickle.UnpicklingError:  
 self.parent.show\_error("File was invalid.")  
 *return  
  
 try*:  
 self.parent.server.thread\_stop(self.parent.server\_thread)  
 *except*:  
 *# The server hasn't been set up, any error doesn't matter  
 # because a new ones being made anyway.  
 pass* self.parent.dht = loaded\_dht  
  
 *try*:  
 self.parent.server = networking.TCPServer(self.parent.dht.node)  
 self.parent.server\_thread = self.parent.server.thread\_start()  
 self.parent.make\_network\_frame()  
 *except* Exception *as* e:  
 self.parent.show\_error(str(e))  
 *return  
  
  
class* JoinNetworkMenuFrame(ctk.CTkFrame):  
 *"""  
 └── Join  
 ├── Settings  
 ├── Load an existing network  
 ├── Create new network  
 └── Bootstrap into a new network  
 """  
  
 def* \_\_init\_\_(self, parent: MainGUI, fg\_color="transparent", \*\*kwargs):  
 ctk.CTkFrame.\_\_init\_\_(self, parent, \*\*kwargs)  
  
 self.configure(fg\_color=fg\_color)  
 self.parent = parent  
  
 join\_title = ctk.CTkLabel(master=self, text="Join Network", font=Fonts.title\_font)  
 join\_title.pack(padx=50, pady=20)  
  
 self.load\_button = ctk.CTkButton(master=self, text="Join stored network", font=Fonts.text\_font,  
 command=self.parent.make\_load\_dht\_frame)  
 self.load\_button.pack(padx=50, pady=10)  
  
 self.bootstrap\_button = ctk.CTkButton(master=self, text="Bootstrap into existing network", font=Fonts.text\_font,  
 command=self.parent.make\_bootstrap\_frame)  
 self.bootstrap\_button.pack(padx=50, pady=10)  
  
 self.create\_new\_network\_button = ctk.CTkButton(master=self, text="Create new network", font=Fonts.text\_font,  
 command=self.parent.initialise\_kademlia)  
 self.create\_new\_network\_button.pack(padx=50, pady=10)  
  
  
*class* BootstrapFromJSONFrame(ctk.CTkFrame):  
 *def* \_\_init\_\_(self, parent: MainGUI, fg\_color="transparent", \*\*kwargs):  
 ctk.CTkFrame.\_\_init\_\_(self, parent, \*\*kwargs)  
 self.configure(fg\_color=fg\_color)  
 self.parent = parent  
  
 self.title = ctk.CTkLabel(self, text="Bootstrap from Contact JSON", font=Fonts.title\_font)  
 self.title.grid(row=0, column=0, columnspan=2, padx=10, pady=20)  
  
 filename\_label = ctk.CTkLabel(master=self, text="Filename:", font=Fonts.text\_font)  
 filename\_label.grid(row=1, column=0, padx=5, pady=5, sticky="nsew")  
  
 self.filename\_entry = ctk.CTkEntry(master=self, width=150, font=Fonts.text\_font)  
 self.filename\_entry.grid(row=1, column=1, padx=5, pady=5, sticky="ew")  
  
 self.back\_button = ctk.CTkButton(master=self, text="Back", font=Fonts.text\_font,  
 command=self.parent.make\_join\_dht\_frame)  
 self.back\_button.grid(row=2, column=0, columnspan=1, padx=10, pady=10)  
  
 self.load\_button = ctk.CTkButton(master=self, text="Load", font=Fonts.text\_font,  
 command=self.load\_known\_peer\_json\_for\_bootstrap)  
 self.load\_button.grid(row=2, column=1, columnspan=1, padx=10, pady=10)  
  
 *def* load\_known\_peer\_json\_for\_bootstrap(self):  
 filename = self.filename\_entry.get().strip("\n")  
 *if not* exists(filename):  
 self.parent.show\_error("Couldn't find file to bootstrap from.")  
 *else*:  
 *with* open(filename, "r") *as* f:  
 contact\_dict = json.load(f)  
  
 known\_id = *None  
 if* "id" *in* contact\_dict:  
 known\_id = contact\_dict["id"]  
 *else*:  
 self.parent.show\_error("File to bootstrap from had no \nparameter 'id'.")  
  
 known\_url = *None  
 if* "url" *in* contact\_dict:  
 known\_url = contact\_dict["url"]  
 *else*:  
 self.parent.show\_error("File to bootstrap from had no \nparameter 'url'.")  
  
 known\_port = *None  
 if* "port" *in* contact\_dict:  
 known\_port = contact\_dict["port"]  
 *else*:  
 self.parent.show\_error("File to bootstrap from had no \nparameter 'port'.")  
  
 *if* known\_url *and* known\_port *and* known\_url:  
 success\_bootstrapping: bool = BootstrapFrame.bootstrap(  
 parent=self.parent,  
 known\_id=id.ID(known\_id),  
 known\_url=known\_url,  
 known\_port=known\_port  
 )  
 *if* success\_bootstrapping:  
 self.parent.make\_network\_frame()  
  
  
*class* BootstrapFrame(ctk.CTkFrame):  
 *def* \_\_init\_\_(self, parent: MainGUI, fg\_color="transparent", \*\*kwargs):  
  
 ctk.CTkFrame.\_\_init\_\_(self, parent, \*\*kwargs)  
 self.configure(fg\_color=fg\_color)  
 self.parent = parent  
  
 self.title = ctk.CTkLabel(self, text="Bootstrap from known peer", font=Fonts.title\_font)  
 self.title.grid(row=0, column=0, columnspan=2, padx=10, pady=20)  
  
 ip\_text = ctk.CTkLabel(master=self, text="IP Address: ")  
 ip\_text.grid(row=1, column=0, padx=5, pady=10, sticky="nsew")  
  
 self.ip\_entry = ctk.CTkEntry(master=self, width=150)  
 self.ip\_entry.grid(row=1, column=1, padx=5, pady=10, sticky="ew")  
  
 port\_text = ctk.CTkLabel(master=self, text="Port: ")  
 port\_text.grid(row=2, column=0, padx=5, pady=10, sticky="nsew")  
  
 self.port\_entry = ctk.CTkEntry(master=self, width=150)  
 self.port\_entry.grid(row=2, column=1, padx=5, pady=10, sticky="ew")  
  
 id\_text = ctk.CTkLabel(master=self, text="ID: ")  
 id\_text.grid(row=3, column=0, padx=5, pady=10, sticky="nsew")  
  
 self.id\_entry = ctk.CTkEntry(master=self, width=150)  
 self.id\_entry.grid(row=3, column=1, padx=5, pady=10, sticky="ew")  
  
 self.back\_button = ctk.CTkButton(self, text="Back", font=Fonts.text\_font,  
 command=self.parent.make\_join\_dht\_frame)  
 self.back\_button.grid(row=4, column=0, columnspan=1, padx=5, pady=10)  
  
 self.load\_from\_json\_button = ctk.CTkButton(self, text="Load from file", font=Fonts.text\_font,  
 command=self.parent.make\_bootstrap\_from\_json\_frame)  
 self.load\_from\_json\_button.grid(row=4, column=1, columnspan=1, padx=5, pady=10)  
  
 self.connect\_button = ctk.CTkButton(master=self, text="Connect", font=Fonts.text\_font,  
 command=self.handle\_bootstrap)  
 self.connect\_button.grid(row=5, column=0, columnspan=2, padx=5, pady=10)  
  
 *def* handle\_bootstrap(self):  
 valid = *False* known\_ip: str = self.ip\_entry.get().strip("\n")  
 ip\_regex = r"(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])"  
 *if not* known\_ip:  
 self.parent.show\_error("IP address must not be empty.")  
 *elif not* re.match(string=known\_ip, pattern=ip\_regex):  
 self.parent.show\_error("IP address is invalid.")  
 *else*:  
 valid = *True* known\_port\_str: str = self.port\_entry.get().strip("\n")  
 known\_port = *None  
 if not* known\_port\_str:  
 self.parent.show\_error("Port must not be empty.")  
 *elif not* known\_port\_str.isnumeric():  
 self.parent.show\_error("Port was not a number.")  
 *elif* int(known\_port\_str) < 0 *or* int(known\_port\_str) > 65535:  
 self.parent.show\_error("Port was out of range. Must be between 0 and 65535.")  
 *else*:  
 known\_port = int(known\_port\_str)  
 valid = *True* known\_id\_value: str = self.id\_entry.get().strip("\n")  
 known\_id = *None  
 if not* known\_id\_value:  
 self.parent.show\_error("ID must not be empty.")  
 *elif not* known\_id\_value.isnumeric():  
 self.parent.show\_error("ID was not a number.")  
 *elif* int(known\_id\_value) < 0 *or* int(known\_id\_value) >= 2 \*\* Constants.ID\_LENGTH\_BITS:  
 *# what if they want to change ID range?* self.parent.show\_error("ID out of range")  
 *else*:  
 known\_id = id.ID(int(known\_id\_value))  
 valid = *True  
  
 if* known\_id *and* known\_ip *and* known\_port *and* valid:  
 self.bootstrap(self.parent, known\_id, known\_ip, known\_port)  
  
 @classmethod  
 *def* bootstrap(cls, parent: MainGUI, known\_id: id.ID, known\_url: str, known\_port: int) -> bool:  
 *"""Attempts to bootstrap Kademlia connection from a known contact"""* known\_protocol = protocols.TCPProtocol(  
 url=known\_url, port=known\_port  
 )  
  
 known\_contact: contact.Contact = contact.Contact(  
 id=known\_id,  
 protocol=known\_protocol  
 )  
 print("[GUI] Bootstrapping from known contact")  
 *if not* hasattr(parent, "dht"):  
 parent.initialise\_kademlia()  
 *return True* print("[GUI] Connecting to known peer's network...")  
 *try*:  
 parent.dht.bootstrap(known\_contact)  
 print("[GUI] Connected to known peer's network.")  
 *return True  
  
 except* errors.RPCError *as* e:  
 *if* e.timeout\_error:  
 parent.show\_error("Timeout error trying to contact known peer.")  
 *return False  
 elif* e.id\_mismatch\_error:  
 parent.show\_error("Random ID returned does not match what was sent.")  
 *return False  
 elif* e.peer\_error:  
 parent.show\_error(f"Peer error: {e}")  
 *return False  
 elif* e.protocol\_error:  
 parent.show\_error(f"Protocol error: {e}")  
 *return False  
 except* Exception *as* e:  
 parent.show\_error(str(e))  
 print(f"[ERROR] Error bootstrapping: {e}")  
 *return False  
  
  
if* \_\_name\_\_ == "\_\_main\_\_":  
 app = MainGUI("dark") *# can also be light* app.mainloop()  
 print("Done!")  
 exit(0)

## Unit tests

*import* math  
*import* os  
*import* random  
*import* shutil  
*import* unittest  
  
*from* kademlia.buckets *import* BucketList, KBucket  
*from* kademlia.constants *import* Constants  
*from* kademlia.contact *import* Contact  
*from* kademlia.dht *import* DHT  
*from* kademlia.errors *import* RPCError, TooManyContactsError  
*from* kademlia.id *import* ID  
*from* kademlia.networking *import* TCPSubnetServer  
*from* kademlia.node *import* Node  
*from* kademlia.protocols *import* TCPSubnetProtocol, VirtualProtocol  
*from* kademlia.routers *import* ParallelRouter, Router  
*from* kademlia.storage *import* VirtualStorage, SecondaryJSONStorage  
  
  
*def* setup\_split\_failure(bucket\_list=*None*):  
 *# force host node ID to < 2 \*\* 159 so the node ID is not in the  
 # 2 \*\* 159 ... 2 \*\* 160 range.  
  
 # b\_host\_id = bytearray()  
 # b\_host\_id.extend([0] \* 20)  
 # b\_host\_id[19] = 0x7F  
  
 # May be incorrect - book does some weird byte manipulation.* host\_id: ID = ID.random\_id(2 \*\* 158, 2 \*\* 159 - 1)  
  
 dummy\_contact: Contact = Contact(host\_id, VirtualProtocol())  
 dummy\_contact.protocol.node = Node(dummy\_contact, VirtualStorage())  
  
 *if not* bucket\_list:  
 bucket\_list = BucketList(our\_contact=dummy\_contact)  
 bucket\_list.our\_id = host\_id  
  
 *# Also add a contact in this 0 - 2 \*\* 159 range  
 # This ensures that only 1 bucket split will occur after 20 nodes with ID >= 2 \*\* 159 are added.* dummy\_contact = Contact(ID(1), VirtualProtocol())  
 dummy\_contact.protocol.node = Node(dummy\_contact, VirtualStorage())  
 bucket\_list.add\_contact(Contact(ID(1), dummy\_contact.protocol))  
  
 *assert* len(bucket\_list.buckets) == 1 *# Bucket split should not have occurred.  
 assert* len(bucket\_list.buckets[0].contacts) == 1 *# Expected 1 contact in bucket 0.  
  
 # Make sure contact IDs all have the same 5-bit prefix and are   
 # in the 2 \*\* 159 ... 2 \*\* 160 - 1 space.* b\_contact\_id: bytearray = bytearray()  
 b\_contact\_id.extend([0] \* 20)  
 b\_contact\_id[19] = 0x80  
  
 *# 1000 xxxx prefix, xxxx starts at 1000 (8)  
 # this ensures that all the contacts in a bucket match only the   
 # prefix as only the first 5 bits are shared.  
 # |----| shared range  
 # 1000 1000 ...  
 # 1000 1100 ...  
 # 1000 1110 ...* shifter = 0x08  
 pos: int = 19  
  
 *for* \_ *in* range(Constants.K):  
 b\_contact\_id[pos] |= shifter *# |= is Bitwise OR.* contact\_id: ID = ID(  
 int.from\_bytes(b\_contact\_id, byteorder="little")  
 )  
 dummy\_contact = Contact(ID(1), VirtualProtocol())  
 dummy\_contact.protocol.node = Node(dummy\_contact, VirtualStorage())  
 bucket\_list.add\_contact(  
 Contact(contact\_id, dummy\_contact.protocol)  
 )  
 shifter >>= 1 *# Right shift (Halves the shift value.)  
 if* shifter == 0:  
 shifter = 0x80  
 pos -= 1  
 *return* bucket\_list  
  
  
*class* KBucketTest(unittest.TestCase):  
  
 *def* test\_add\_to\_kbucket(*self*):  
 *"""  
 Description  
 Contact added to a full bucket.  
  
 Expected  
 “TooManyContactsError” should be raised.  
  
 :return:  
 """* k = Constants.K  
 k\_bucket = KBucket()  
 *for* i *in* range(k):  
 contact = Contact(ID(i))  
 k\_bucket.add\_contact(contact)  
 *self*.assertTrue(len(k\_bucket.contacts) == k)  
  
 *def* test\_too\_many\_contacts(*self*):  
 *"""  
 Description  
 Contact added to an almost full bucket.  
  
 Expected  
 No exceptions should be raised, length of bucket contacts should be Constants.K  
  
 Notes  
 This implies standard behaviour of contacts being added to a relatively empty bucket,  
 as it is required as a prerequisite to this test.  
 :return:  
 """* k = Constants.K  
 k\_bucket = KBucket()  
 *for* i *in* range(k):  
 contact = Contact(ID(i))  
 k\_bucket.add\_contact(contact)  
 *with self*.assertRaises(TooManyContactsError):  
 *# Trying to add one more contact should raise the exception* contact = Contact(ID(k + 1))  
 k\_bucket.add\_contact(contact)  
  
 *def* test\_no\_funny\_business(*self*):  
 *"""  
 Description  
 Compare a Kbucket with no initial contacts parameter, to one  
 with an empty initial contacts parameter  
  
 Expected  
 They are the same.*<https://www.msn.com/en-gb/money/watchlist?id=aop6h7&tab=Markets&ocid=winp2fptaskbarhoverent&cvid=f3f1a830e42e481bf2276b4a177907a0> *:return:  
 """* k1: KBucket = KBucket(low=0, high=100)  
 k2: KBucket = KBucket(low=10, high=200, initial\_contacts=[])  
 *self*.assertTrue(k1.contacts == k2.contacts)  
  
  
*class* AddContactTest(unittest.TestCase):  
  
 *def* test\_unique\_id\_add(*self*):  
 *"""  
 Description  
  
 Adding K contacts to bucket list.  
  
 Expected  
  
 Bucket list should not split into separate buckets, and K contacts should exist in one bucket.  
 :return:  
 """* dummy\_contact: Contact = Contact(id=ID(0),  
 protocol=VirtualProtocol())  
  
 dummy\_contact.protocol.node = Node(dummy\_contact, VirtualStorage())  
  
 bucket\_list: BucketList = BucketList(dummy\_contact)  
 bucket\_list.our\_id = ID.random\_id()  
  
 *for* i *in* range(Constants.K):  
 bucket\_list.add\_contact(Contact(ID.random\_id()))  
  
 *self*.assertTrue(  
 len(bucket\_list.buckets) == 1, "No split should have taken place.")  
  
 *self*.assertTrue(  
 len(bucket\_list.buckets[0].contacts) == Constants.K,  
 "K contacts should have been added.")  
  
 *def* test\_duplicate\_id(*self*):  
 *"""  
 Description  
  
 Adding 1 contact to a bucket list twice.  
  
 Expected  
 The bucket list should realise that the contact ID already exists in the buckets,  
 therefore it should not be added.  
 :return:  
 """* dummy\_contact = Contact(ID(0), VirtualProtocol())  
 dummy\_contact.protocol.node = Node(dummy\_contact, VirtualStorage())  
 bucket\_list: BucketList = BucketList(dummy\_contact)  
 bucket\_list.our\_id = ID.random\_id()  
  
 id: ID = ID.random\_id()  
  
 bucket\_list.add\_contact(Contact(id))  
 bucket\_list.add\_contact(Contact(id))  
  
 *self*.assertTrue(  
 len(bucket\_list.buckets) == 1, "No split should have taken place.")  
  
 *self*.assertTrue(  
 len(bucket\_list.buckets[0].contacts) == 1,  
 "Bucket should have one contact.")  
  
 *def* test\_bucket\_split(*self*):  
 *"""  
 Description  
  
 Adding K + 1 contacts to an empty bucket list.  
  
 Expected  
  
 The bucket list should split into 2 separate buckets.  
  
  
 :return:  
 """* dummy\_contact = Contact(ID(0), VirtualProtocol())  
 dummy\_contact.protocol.node = Node(dummy\_contact, VirtualStorage())  
 bucket\_list: BucketList = BucketList(dummy\_contact)  
 bucket\_list.our\_id = ID.random\_id()  
 *for* i *in* range(Constants.K):  
 bucket\_list.add\_contact(Contact(ID.random\_id()))  
 bucket\_list.add\_contact(Contact(ID.random\_id()))  
  
 print(f"KBucket range for first bucket: {bucket\_list.buckets[0].low()}, "  
 f"{bucket\_list.buckets[0].high()}, high log 2: {math.log(bucket\_list.buckets[0].high(), 2)}")  
 print(f"KBucket range for second bucket: {bucket\_list.buckets[1].low()}, "  
 f"{bucket\_list.buckets[1].high()}, high log 2: {math.log(bucket\_list.buckets[1].high(), 2)}")  
  
 *self*.assertTrue(  
 len(bucket\_list.buckets) > 1,  
 "Bucket should have split into two or more buckets. "  
 f"Length of first buckets contacts = {len(bucket\_list.buckets[0].contacts)}")  
  
  
*class* ForceFailedAddTest(unittest.TestCase):  
 *def* test\_force\_failed\_add(*self*):  
 *"""  
 Description  
  
 Creates a bucket list composed of K ID’s, with a depth of 5 in the range 2 \*\* 159 to 2 \*\* 160 – 1,  
 along with another Contact with ID in range 0 to 2 \*\* 159 – 1.  
 Then another contact should be added with ID >= 2 \*\* 159.  
  
 Expected  
  
 Bucket split should occur, with 1 contact in the first bucket, and 20 contacts in the second bucket.  
 Then when the 22nd contact is added, nothing should have changed, due to the depth of the bucket it’s being  
 added to MOD 5 is 0.  
 :return:  
 """* dummy\_contact = Contact(id=ID(0))  
 node = Node(contact=dummy\_contact, storage=VirtualStorage())  
  
 bucket\_list: BucketList = setup\_split\_failure()  
  
 *self*.assertTrue(len(bucket\_list.buckets) == 2,  
 f"Bucket split should have occurred. Number of buckets should be 2, is {len(bucket\_list.buckets)}.")  
  
 *self*.assertTrue(len(bucket\_list.buckets[0].contacts) == 1,  
 "Expected 1 contact in bucket 0.")  
  
 *self*.assertTrue(len(bucket\_list.buckets[1].contacts) == 20,  
 "Expected 20 contacts in bucket 1.")  
  
 *# This next contact should not split the bucket as  
 # depth == 5 and therefore adding the contact will fail.  
  
 # Any unique ID >= 2^159 will do.* id = 2 \*\* 159 + 4  
  
 new\_contact = Contact(id=ID(id),  
 protocol=dummy\_contact.protocol)  
 bucket\_list.add\_contact(new\_contact)  
  
 *self*.assertTrue(len(bucket\_list.buckets) == 2,  
 f"Bucket split should have occured. Number of buckets should be 2, is {len(bucket\_list.buckets)}.")  
  
 *self*.assertTrue(len(bucket\_list.buckets[0].contacts) == 1,  
 "Expected 1 contact in bucket 0.")  
  
 *self*.assertTrue(len(bucket\_list.buckets[1].contacts) == 20,  
 "Expected 20 contacts in bucket 1.")  
  
 *self*.assertTrue(new\_contact *not in* bucket\_list.buckets[1].contacts,  
 "Expected new contact NOT to replace an older contact.")  
  
  
*# 3 Here don't work  
class* NodeLookupTests(unittest.TestCase):  
  
 *def* test\_get\_close\_contacts\_ordered(*self*):  
 *"""  
 Description  
  
 Adds 100 random contacts to a nodes bucket list, then FIND\_NODE is performed.  
  
 Expected  
  
 K Contacts should be returned; Returned contacts should be ordered by distance.  
 It should have returned the smallest ID’s possible, as host ID = 0.  
  
 :return:  
 """* sender: Contact = Contact(id=ID.random\_id(),  
 protocol=*None*)  
 node: Node = Node(  
 Contact(id=ID.random\_id(), protocol=*None*),  
 VirtualStorage())  
  
 contacts: list[Contact] = []  
 *for* \_ *in* range(100):  
 contacts.append(  
 Contact(id=ID.random\_id(), protocol=*None*))  
  
 *for* contact *in* contacts:  
 node.bucket\_list.add\_contact(contact)  
  
 key: ID = ID.random\_id()  
  
 closest: list[Contact] = node.find\_node(sender=sender, key=key)[0]  
 *# print(closest)  
 self*.assertTrue(len(closest) == Constants.K,  
 "Expected K contacts to be returned.")  
  
 *# the contacts are already in ascending order with respect to the key.* distances: list[int] = [c.id ^ key *for* c *in* closest]  
 distance: int = distances[0]  
  
 *# checking they're all in order (ascending)  
 for* i *in* distances[1:]:  
 *self*.assertTrue(distance < i,  
 "Expected contacts to be ordered by distance.")  
 distance = i  
  
 *# Verify the contacts with the smallest distances have been returned from all possible distances.* last\_distance = distances[-1]  
  
 *# This just makes sure it returned the K smallest contact ID's possible.* others = []  
 *for* b *in* node.bucket\_list.buckets:  
 *for* c *in* b.contacts:  
 *if* c *not in* closest *and* (c.id ^ key) < last\_distance:  
 others.append(c)  
  
 *self*.assertTrue(  
 len(others) == 0,  
 "Expected no other contacts with a smaller distance than the greatest distance to exist."  
 )  
  
 *def* test\_no\_nodes\_to\_query(*self*):  
 *"""  
 Creates K nodes and adds them to our routers bucket list, where each node knows about  
 the other peers.  
 :return:  
 """* router\_node\_contact = Contact(id=ID.random\_id(),  
 protocol=*None*)  
 router = Router(  
 node=Node(contact=router\_node\_contact, storage=VirtualStorage()))  
  
 nodes: list[Node] = []  
  
 *for* i *in* range(Constants.K):  
 nodes.append(  
 Node(Contact(id=ID(2 \*\* i)), storage=VirtualStorage()))  
  
 *for* n *in* nodes:  
 *# fixup protocols* n.our\_contact.protocol = VirtualProtocol(n)  
  
 *# our contacts:* router.node.bucket\_list.add\_contact(n.our\_contact)  
  
 *# each peer needs to know about the other peers* n\_other = [i *for* i *in* nodes *if* i *is not* n] *# MIGHT ERROR  
 # n\_other = [i for i in nodes if i != n]  
  
 # From book:  
 # nodes.ForEach(n => nodes.Where(nOther => nOther != n).  
 # ForEach(nOther => n.BucketList.AddContact(nOther.OurContact)));  
 for* other\_node *in* n\_other:  
 n.bucket\_list.add\_contact(other\_node.our\_contact)  
  
 *# select the key such that n^0==n* key = ID(0)  
 *# all contacts are in one bucket (?)* contacts\_to\_query = router.node.bucket\_list.buckets[0].contacts  
 closer\_contacts: list[Contact] = []  
 further\_contacts: list[Contact] = []  
  
 *for* c *in* contacts\_to\_query:  
 *# should I read the output?* found, val, found\_by, closer\_contacts, further\_contacts = \  
 router.get\_closer\_nodes(key=key,  
 node\_to\_query=c,  
 rpc\_call=router.rpc\_find\_nodes,  
 closer\_contacts=closer\_contacts,  
 further\_contacts=further\_contacts)  
  
 closer\_compare\_arr = []  
 *for* contact *in* further\_contacts:  
 *if* contact.id *not in* [i.id *for* i *in* contacts\_to\_query]:  
 closer\_compare\_arr.append(contact)  
  
 *self*.assertTrue(len(closer\_compare\_arr) == 0, "No new nodes expected.")  
  
 further\_compare\_arr = []  
 *for* contact *in* further\_contacts:  
 *if* contact.id *not in* [i.id *for* i *in* contacts\_to\_query]:  
 further\_compare\_arr.append(contact)  
  
 *self*.assertTrue(len(further\_compare\_arr) == 0, "No new nodes expected.")  
  
 *def* \_\_setup(*self*):  
 *self*.router = Router(  
 Node(Contact(id=ID.random\_id(), protocol=*None*),  
 storage=VirtualStorage()))  
  
 *self*.nodes: list[Node] = []  
 *for* \_ *in* range(100):  
 contact: Contact = Contact(id=ID.random\_id(), protocol=VirtualProtocol())  
 node: Node = Node(contact, VirtualStorage())  
 contact.protocol.node = node  
 *self*.nodes.append(node)  
  
 *for* n *in self*.nodes:  
 n.our\_contact.protocol = VirtualProtocol(n) *# Fix up protocols  
 self*.router.node.bucket\_list.add\_contact(n.our\_contact)  
 *for* other\_n *in self*.nodes: *# let each node know about each other node  
 if* other\_n != n:  
 n.bucket\_list.add\_contact(other\_n.our\_contact)  
  
 *# pick a random bucket* key = ID.random\_id()  
 *# take "A" contacts from a random KBucket  
 self*.contacts\_to\_query: list[Contact] = \  
 *self*.router.node.bucket\_list.get\_kbucket(key).contacts[:Constants.A]  
  
 *self*.closer\_contacts: list[Contact] = []  
 *self*.further\_contacts: list[Contact] = []  
  
 *self*.closer\_contacts\_alt\_computation: list[Contact] = []  
 *self*.further\_contacts\_alt\_computation: list[Contact] = []  
  
 *self*.nearest\_contact\_node = sorted(*self*.contacts\_to\_query,  
 key=*lambda* contacts\_to\_query\_nodes: contacts\_to\_query\_nodes.id ^ key)[0]  
 *self*.distance = *self*.nearest\_contact\_node.id ^ key  
  
 *def* get\_alt\_close\_and\_far(*self*, contacts\_to\_query: list[Contact],  
 closer: list[Contact],  
 further: list[Contact],  
 nodes: list[Node],  
 key: ID, *# I think this is needed.* distance  
 ):  
 *"""  
 Alternate implementation for getting closer and further contacts.  
 """  
 # For each node (A == K) for testing in our bucket (nodes\_to\_query  
 for* contact *in* contacts\_to\_query:  
 *# Find the node that we're contacting:* contact\_node: Node = next((n *for* n *in* nodes *if* n.our\_contact == contact), *None*)  
 *if* contact\_node *is None*:  
 *continue  
  
 # Close contacts except ourself and the nodes we're contacting.  
 # Note that of all the contacts in the bucket list, many of the K returned  
 # by the get\_close\_contacts call are contacts we're querying, so they're being excluded.* close\_contacts\_of\_contacted\_node = [  
 c *for* c *in* contact\_node.bucket\_list.get\_close\_contacts(key, *self*.router.node.our\_contact.id)  
 *if* c.id.value *not in* [c.id.value *for* c *in* contacts\_to\_query]  
 ]  
  
 *for* close\_contact\_of\_contacted\_node *in* close\_contacts\_of\_contacted\_node:  
 *# Which of these contacts are closer?  
 if* (close\_contact\_of\_contacted\_node.id.value ^ key.value < distance *and* close\_contact\_of\_contacted\_node.id.value *not in* [c.id.value *for* c *in* closer]):  
 closer.append(close\_contact\_of\_contacted\_node)  
  
 *# Which of these contacts are farther?  
 if* close\_contact\_of\_contacted\_node.id.value ^ key.value >= distance *and* close\_contact\_of\_contacted\_node.id.value *not in* [  
 c.id.value *for* c *in* further]:  
 further.append(close\_contact\_of\_contacted\_node)  
  
 *def* test\_lookup(*self*):  
  
 *for* i *in* range(100):  
 id = ID.random\_id(seed=i)  
  
 *self*.\_\_setup()  
  
 close\_contacts: list[Contact] = *self*.router.lookup(  
 key=id, rpc\_call=*self*.router.rpc\_find\_nodes, give\_me\_all=*True*)["contacts"]  
  
 contacted\_nodes: list[Contact] = close\_contacts  
 *self*.get\_alt\_close\_and\_far(*self*.contacts\_to\_query,  
 *self*.closer\_contacts\_alt\_computation,  
 *self*.further\_contacts\_alt\_computation,  
 *self*.nodes,  
 key=id,  
 distance=*self*.distance)  
  
 print("close\_contacts", [str(i.id.value)[-3:] *for* i *in* close\_contacts])  
 print("closer\_contacts\_alt\_computation",  
 [str(i.id.value)[-3:] *for* i *in self*.closer\_contacts\_alt\_computation])  
 print("\n\n")  
  
 *# Check closer\_contacts\_alt\_computation is right:  
 for* contact *in self*.closer\_contacts\_alt\_computation:  
 print("Distance delta close\_alt:", ((contact.id.value ^ id.value) - *self*.distance) < 0,  
 ((contact.id.value ^ id.value) - *self*.distance))  
  
 *for* contact *in* close\_contacts:  
 print("Distance delta close\_contacts:", ((contact.id.value ^ id.value) - *self*.distance) < 0,  
 ((contact.id.value ^ id.value) - *self*.distance))  
  
 *self*.assertTrue(len(close\_contacts) >= len(*self*.closer\_contacts\_alt\_computation),  
 f"Expected at least as many contacts: {len(close\_contacts)} vs {len(*self*.closer\_contacts\_alt\_computation)}")  
  
 *for* c *in self*.closer\_contacts\_alt\_computation:  
 *self*.assertTrue(c *in* close\_contacts,  
 "somehow a close contact in the computation is not in the originals?")  
  
 *def* test\_simple\_all\_closer\_contacts(*self*):  
 *# setup  
 # by selecting our node ID to zero, we ensure that all distances of other nodes   
 # are greater than the distance to our node.  
  
 # Create a router with the largest ID possible.* router = Router(Node(Contact(id=ID.max(), protocol=*None*), VirtualStorage()))  
 nodes: list[Node] = []  
  
 *for* n *in* range(Constants.K):  
 *# Create a node with id of a power of 2, up to 2\*\*20.* node = Node(Contact(id=ID(2 \*\* n), protocol=*None*), storage=VirtualStorage())  
 nodes.append(node)  
  
 *# Fixup protocols  
 for* n *in* nodes:  
 n.our\_contact.protocol = VirtualProtocol(n)  
  
 *# add all contacts in our node list to the router.  
 for* n *in* nodes:  
 router.node.bucket\_list.add\_contact(n.our\_contact)  
  
 *# let all of them know where the others are:  
 # (add each nodes contact to each nodes bucket\_list)  
 for* n *in* nodes:  
 *for* n\_other *in* nodes:  
 *if* n != n\_other:  
 n.bucket\_list.add\_contact(n\_other.our\_contact)  
  
 *# select the key such that n ^ 0 == n (TODO: Why?)  
 # this ensures the distance metric uses only the node ID,  
 # which makes for an integer difference for distance, not an XOR distance.* key = ID(0)  
 *# all contacts are in one bucket  
 # This is because we added K node's contacts to router,  
 # so it shouldn't have split.* contacts\_to\_query = router.node.bucket\_list.buckets[0].contacts  
  
 contacts = router.lookup(key=key,  
 rpc\_call=router.rpc\_find\_nodes,  
 give\_me\_all=*True*)  
  
 *# Make sure lookup returns K contacts.  
 self*.assertTrue(len(contacts) == Constants.K, f"Expected K closer contacts, got {len(contacts)}.")  
  
 *# Make sure it realises all contacts should be closer than 2\*\*160 - 1.  
 self*.assertTrue(len(router.closer\_contacts) == Constants.K,  
 "All contacts should be closer than the ID 2\*\*160 - 1.")  
  
 *self*.assertTrue(len(router.further\_contacts) == 0, "Expected no further contacts.")  
  
 *def* test\_simple\_all\_further\_contacts(*self*):  
 *# setup  
 # by selecting our node ID to zero, we ensure that all distances of other nodes   
 # are greater than the distance to our node.  
  
 # Create a router with the largest ID possible.* router = Router(Node(Contact(id=ID(0), protocol=*None*), VirtualStorage()))  
 nodes: list[Node] = []  
  
 *for* n *in* range(Constants.K):  
 *# Create a node with id of a power of 2, up to 2\*\*20.* node = Node(Contact(id=ID(2 \*\* n), protocol=*None*), storage=VirtualStorage())  
 nodes.append(node)  
  
 *# Fixup protocols  
 for* n *in* nodes:  
 n.our\_contact.protocol = VirtualProtocol(n)  
  
 *# add all contacts in our node list to the router.  
 for* n *in* nodes:  
 router.node.bucket\_list.add\_contact(n.our\_contact)  
  
 *# let all of them know where the others are:  
 # (add each nodes contact to each nodes bucket\_list)  
 for* n *in* nodes:  
 *for* n\_other *in* nodes:  
 *if* n != n\_other:  
 n.bucket\_list.add\_contact(n\_other.our\_contact)  
  
 *# select the key such that n ^ 0 == n  
 # this ensures the distance metric uses only the node ID,  
 # which makes for an integer difference for distance, not an XOR distance.* key = ID(0)  
 *# all contacts are in one bucket  
 # This is because we added K node's contacts to router,  
 # so it shouldn't have split.* contacts\_to\_query = router.node.bucket\_list.buckets[0].contacts  
  
 contacts = router.lookup(key=key,  
 rpc\_call=router.rpc\_find\_nodes,  
 give\_me\_all=*True*)  
  
 *# Make sure lookup returns K contacts.  
 self*.assertTrue(len(contacts) == Constants.K, f"Expected K closer contacts, got {len(contacts)}.")  
  
 *# Make sure it realises all contacts should be further than the ID 0.  
 self*.assertTrue(len(router.further\_contacts) == Constants.K,  
 "All contacts should be further than the ID 0.")  
  
 *self*.assertTrue(len(router.closer\_contacts) == 0, "Expected no closer contacts.")  
  
  
*class* DHTTest(unittest.TestCase):  
 *def* test\_local\_store\_find\_value(*self*):  
 vp = VirtualProtocol()  
 *# Below line should contain VirtualStorage(), which I don't have?* dht = DHT(id=ID.random\_id(),  
 protocol=vp,  
 storage\_factory=VirtualStorage,  
 router=Router())  
 *# print(dht.\_originator\_storage)* vp.node = dht.\_router.node  
 key = ID.random\_id()  
 dht.store(key, "Test")  
 found, contacts, return\_val = dht.find\_value(key)  
 print(found, contacts, return\_val)  
 *self*.assertTrue(return\_val == "Test",  
 "Expected to get back what we stored.")  
  
 *def* test\_value\_stored\_in\_closer\_node(*self*):  
 *"""  
 This test creates a single contact and stores the value in that contact. We set up the IDs so that the  
 contact’s ID is less (XOR metric) than our peer’s ID, and we use a key of ID.Zero to prevent further  
 complexities when computing the distance. Most of the code here is to set up the conditions to make this test!  
 - "The Kademlia Protocol Succinctly" by Marc Clifton  
 :return: None  
 """* vp1 = VirtualProtocol()  
 vp2 = VirtualProtocol()  
 store1 = VirtualStorage()  
 store2 = VirtualStorage()  
  
 *# Ensures that all nodes are closer, because id.max ^ n < id.max when n > 0.  
 # (the distance between a node and max id is always closer than the furthest possible)* dht = DHT(id=ID.max(), router=Router(), protocol=vp1, originator\_storage=store1,  
 republish\_storage=store1, cache\_storage=VirtualStorage())  
  
 vp1.node = dht.\_router.node  
 contact\_id: ID = ID.mid() *# middle ID* other\_contact: Contact = Contact(id=contact\_id, protocol=vp2)  
 other\_node = Node(contact=other\_contact, storage=store2)  
 vp2.node = other\_node  
  
 *# add this other contact to our peer list* dht.\_router.node.bucket\_list.add\_contact(other\_contact)  
 *# we want an integer distance, not an XOR distance.* key: ID = ID.min()  
 val = "Test"  
 other\_node.simply\_store(key, val)  
 *self*.assertFalse(store1.contains(key),  
 "Expected our peer to NOT have cached the key-value.")  
  
 *self*.assertTrue(store2.contains(key),  
 "Expected other node to HAVE cached the key-value.")  
  
 *# Try and find the value, given our Dht knows about the other contact.* \_, \_, retval = dht.find\_value(key)  
 *self*.assertTrue(retval == val,  
 "Expected to get back what we stored")  
  
 *def* test\_value\_stored\_in\_further\_node(*self*):  
 vp1 = VirtualProtocol()  
 vp2 = VirtualProtocol()  
 store1 = VirtualStorage()  
 store2 = VirtualStorage()  
  
 *# Ensures that all nodes are closer, because max id ^ n < max id when n > 0.* dht: DHT = DHT(id=ID.min(), protocol=vp1, router=Router(), storage\_factory=*lambda*: store1)  
  
 vp1.node = dht.\_router.node  
 contact\_id = ID.max()  
 other\_contact = Contact(contact\_id, vp2)  
 other\_node = Node(other\_contact, store2)  
 vp2.node = other\_node  
 *# Add this other contact to our peer list.* dht.\_router.node.bucket\_list.add\_contact(other\_contact)  
 key = ID(1)  
 val = "Test"  
 other\_node.simply\_store(key, val)  
  
 *self*.assertFalse(store1.contains(key),  
 "Expected our peer to have NOT cached the key-value.")  
  
 *self*.assertTrue(store2.contains(key),  
 "Expected other node to HAVE cached the key-value.")  
  
 \_, \_, retval = dht.find\_value(key)  
 *self*.assertTrue(retval == val,  
 "Expected to get back what we stored.")  
  
 *def* test\_value\_stored\_gets\_propagated(*self*):  
 vp1 = VirtualProtocol()  
 vp2 = VirtualProtocol()  
 store1 = VirtualStorage()  
 store2 = VirtualStorage()  
  
 dht: DHT = DHT(id=ID.min(), protocol=vp1, router=Router(), storage\_factory=*lambda*: store1)  
 vp1.node = dht.\_router.node  
 contact\_id = ID.mid()  
 other\_contact = Contact(contact\_id, vp2)  
 other\_node = Node(other\_contact, store2)  
 vp2.node = other\_node  
 dht.\_router.node.bucket\_list.add\_contact(other\_contact)  
  
 key = ID(0)  
 val = "Test"  
  
 *self*.assertFalse(store1.contains(key),  
 "Obviously we don't have the key-value yet.")  
  
 *self*.assertFalse(store2.contains(key),  
 "And equally obvious, the other peer doesn't have the key-value yet either.")  
  
 dht.store(key, val)  
  
 *self*.assertTrue(store1.contains(key),  
 "Expected our peer to have stored the key-value.")  
  
 *self*.assertTrue(store2.contains(key),  
 "Expected the other peer to have stored the key-value.")  
  
 *def* test\_value\_propagates\_to\_closer\_node(*self*):  
 vp1 = VirtualProtocol()  
 vp2 = VirtualProtocol()  
 vp3 = VirtualProtocol()  
  
 store1 = VirtualStorage()  
 store2 = VirtualStorage()  
 store3 = VirtualStorage()  
  
 cache3 = VirtualStorage()  
  
 dht: DHT = DHT(id=ID.max(),  
 protocol=vp1,  
 router=Router(),  
 originator\_storage=store1,  
 republish\_storage=store1,  
 cache\_storage=VirtualStorage())  
 vp1.node = dht.\_router.node  
  
 *# setup node 2* contact\_id\_2 = ID.mid()  
 other\_contact\_2 = Contact(contact\_id\_2, vp2)  
 other\_node\_2 = Node(other\_contact\_2, store2)  
 vp2.node = other\_node\_2  
 *# add the second contact to our peer list.* dht.\_router.node.bucket\_list.add\_contact(other\_contact\_2)  
 *# node 2 has the value* key = ID(0)  
 val = "Test"  
 other\_node\_2.storage.set(key, val)  
  
 *# setup node 3* contact\_id\_3 = ID(2 \*\* 158) *# I think this is the same as ID.Zero.SetBit(158)?* print("id", contact\_id\_3)  
 other\_contact\_3 = Contact(contact\_id\_3, vp3)  
 other\_node\_3 = Node(other\_contact\_3, storage=store3, cache\_storage=cache3)  
 vp3.node = other\_node\_3  
 *# add the third contact to our peer list* print("DHT Router DHT", dht.\_router.dht.\_router.dht.\_router.dht.\_router)  
 dht.\_router.node.bucket\_list.add\_contact(other\_contact\_3)  
  
 *self*.assertFalse(store1.contains(key),  
 "Obviously we don't have the key-value yet.")  
  
 *self*.assertFalse(store3.contains(key),  
 "And equally obvious, the third peer doesn't have the key-value yet either.")  
  
 ret\_found, ret\_contacts, ret\_val = dht.find\_value(key)  
  
 *self*.assertTrue(ret\_found, "Expected value to be found.")  
 *self*.assertFalse(store3.contains(key), "Key should not be in the republish store.")  
 *self*.assertTrue(cache3.contains(key), "Key should be in the cache store.")  
 *self*.assertTrue(cache3.get\_expiration\_time\_sec(key.value) == Constants.EXPIRATION\_TIME\_SEC / 2,  
 "Expected 12 hour expiration.")  
  
  
*class* DHTParallelTest(unittest.TestCase):  
 *"""  
 The exact same as DHTTest, but with the asynchronous router instead of the normal router.  
 """  
  
 def* test\_local\_store\_find\_value(*self*):  
 vp = VirtualProtocol()  
 *# Below line should contain VirtualStorage(), which I don't have?* dht = DHT(id=ID.random\_id(),  
 protocol=vp,  
 storage\_factory=VirtualStorage,  
 router=ParallelRouter())  
 vp.node = dht.\_router.node  
 key = ID.random\_id()  
 dht.store(key, "Test")  
 \_, \_, return\_val = dht.find\_value(key)  
  
 *self*.assertTrue(return\_val == "Test",  
 "Expected to get back what we stored.")  
  
 *def* test\_value\_stored\_in\_closer\_node(*self*):  
 *"""  
 This test creates a single contact and stores the value in that contact. We set up the IDs so that the  
 contact’s ID is less (XOR metric) than our peer’s ID, and we use a key of ID.Zero to prevent further  
 complexities when computing the distance. Most of the code here is to set up the conditions to make this test!  
 - "The Kademlia Protocol Succinctly" by Marc Clifton  
 :return: None  
 """* vp1 = VirtualProtocol()  
 vp2 = VirtualProtocol()  
 store1 = VirtualStorage()  
 store2 = VirtualStorage()  
  
 *# Ensures that all nodes are closer, because id.max ^ n < id.max when n > 0.  
 # (the distance between a node and max id is always closer than the furthest possible)* dht = DHT(id=ID.max(), router=ParallelRouter(), storage\_factory=*lambda*: store1, protocol=VirtualProtocol())  
  
 vp1.node = dht.\_router.node  
 contact\_id: ID = ID.mid() *# middle ID* other\_contact: Contact = Contact(id=contact\_id, protocol=vp2)  
 other\_node = Node(contact=other\_contact, storage=store2)  
 vp2.node = other\_node  
  
 *# add this other contact to our peer list* dht.\_router.node.bucket\_list.add\_contact(other\_contact)  
 *# we want an integer distance, not an XOR distance.* key: ID = ID.min()  
 val = "Test"  
 other\_node.simply\_store(key, val)  
 *self*.assertFalse(store1.contains(key),  
 "Expected our peer to NOT have cached the key-value.")  
  
 *self*.assertTrue(store2.contains(key),  
 "Expected other node to HAVE cached the key-value.")  
  
 *# Try and find the value, given our Dht knows about the other contact.* \_, \_, retval = dht.find\_value(key)  
 *self*.assertTrue(retval == val,  
 "Expected to get back what we stored")  
  
 *def* test\_value\_stored\_in\_further\_node(*self*):  
 vp1 = VirtualProtocol()  
 vp2 = VirtualProtocol()  
 store1 = VirtualStorage()  
 store2 = VirtualStorage()  
  
 *# Ensures that all nodes are closer, because max id ^ n < max id when n > 0.* dht: DHT = DHT(id=ID.min(), protocol=vp1, router=ParallelRouter(), storage\_factory=*lambda*: store1)  
  
 vp1.node = dht.\_router.node  
 contact\_id = ID.max()  
 other\_contact = Contact(contact\_id, vp2)  
 other\_node = Node(other\_contact, store2)  
 vp2.node = other\_node  
 *# Add this other contact to our peer list.* dht.\_router.node.bucket\_list.add\_contact(other\_contact)  
 key = ID(1)  
 val = "Test"  
 other\_node.simply\_store(key, val)  
  
 *self*.assertFalse(store1.contains(key),  
 "Expected our peer to have NOT cached the key-value.")  
  
 *self*.assertTrue(store2.contains(key),  
 "Expected other node to HAVE cached the key-value.")  
  
 retval: str = dht.find\_value(key)[2]  
 *self*.assertTrue(retval == val,  
 "Expected to get back what we stored.")  
  
 *def* test\_value\_stored\_gets\_propagated(*self*):  
 vp1 = VirtualProtocol()  
 vp2 = VirtualProtocol()  
 store1 = VirtualStorage()  
 store2 = VirtualStorage()  
  
 dht: DHT = DHT(id=ID.min(), protocol=vp1, router=ParallelRouter(), storage\_factory=*lambda*: store1)  
 vp1.node = dht.\_router.node  
 contact\_id = ID.mid()  
 other\_contact = Contact(contact\_id, vp2)  
 other\_node = Node(other\_contact, store2)  
 vp2.node = other\_node  
 dht.\_router.node.bucket\_list.add\_contact(other\_contact)  
  
 key = ID(0)  
 val = "Test"  
  
 *self*.assertFalse(store1.contains(key),  
 "Obviously we don't have the key-value yet.")  
  
 *self*.assertFalse(store2.contains(key),  
 "And equally obvious, the other peer doesn't have the key-value yet either.")  
  
 dht.store(key, val)  
  
 *self*.assertTrue(store1.contains(key),  
 "Expected our peer to have stored the key-value.")  
  
 *self*.assertTrue(store2.contains(key),  
 "Expected the other peer to have stored the key-value.")  
  
 *# def test\_value\_propagates\_to\_closer\_node(self):  
 # vp1 = VirtualProtocol()  
 # vp2 = VirtualProtocol()  
 # vp3 = VirtualProtocol()  
 #  
 # store1 = VirtualStorage()  
 # store2 = VirtualStorage()  
 # store3 = VirtualStorage()  
 #  
 # cache3 = VirtualStorage()  
 #  
 # dht: DHT = DHT(id=ID.max(), protocol=vp1, router=ParallelRouter(), storage\_factory=lambda: store1)  
 #  
 # vp1.node = dht.\_router.node  
 #  
 # # setup node 2  
 # contact\_id\_2 = ID.mid()  
 # other\_contact\_2 = Contact(contact\_id\_2, vp2)  
 # other\_node\_2 = Node(other\_contact\_2, store2)  
 # vp2.node = other\_node\_2  
 # # add the second contact to our peer list.  
 # dht.\_router.node.bucket\_list.add\_contact(other\_contact\_2)  
 # # node 2 has the value  
 # key = ID(0)  
 # val = "Test"  
 # other\_node\_2.storage.set(key, val)  
 #  
 # # setup node 3  
 # contact\_id\_3 = ID(2 \*\* 158) # I think this is the same as ID.Zero.SetBit(158)?  
 # other\_contact\_3 = Contact(contact\_id\_3, vp3)  
 # other\_node\_3 = Node(other\_contact\_3, store3, cache\_storage=cache3)  
 # vp3.node = other\_node\_3  
 # # add the third contact to our peer list  
 # dht.\_router.node.bucket\_list.add\_contact(other\_contact\_3)  
 #  
 # self.assertFalse(store1.contains(key),  
 # "Obviously we don't have the key-value yet.")  
 #  
 # self.assertFalse(store3.contains(key),  
 # "And equally obvious, the third peer doesn't have the key-value yet either.")  
 #  
 # ret\_found, ret\_contacts, ret\_val = dht.find\_value(key)  
 #  
 # self.assertTrue(ret\_found, "Expected value to be found.")  
 # self.assertFalse(store3.contains(key), "Key should not be in the republish store.")  
 # self.assertTrue(cache3.contains(key), "Key should be in the cache store.")  
 # self.assertTrue(cache3.get\_expiration\_time\_sec(key.value) == Constants.EXPIRATION\_TIME\_SEC / 2,  
 # "Expected 12 hour expiration.")  
  
  
class* BootstrappingTests(unittest.TestCase):  
  
 *def* test\_random\_within\_bucket\_tests(*self*):  
  
 test\_cases: list[tuple[int, int]] = [  
  
 (0, 256), *# 7 bits should be set* (256, 1024), *# 2 bits (256 + 512) should be set* (65536, 65536 \* 2), *# no additional bits should be set.* (65536, 65536 \* 4), *# 2 bits (65536 and 65536\*2) should be set.* (65536, 65536 \* 16) *# 4 bits (65536, 65536\*2, 65536\*4, 65536\*8) set.* ]  
 *for* test\_case *in* test\_cases:  
 low = test\_case[0]  
 high = test\_case[1]  
 bucket: KBucket = KBucket(low=low, high=high)  
  
 id: ID = ID.random\_id\_within\_bucket\_range(bucket)  
 *self*.assertTrue(bucket.is\_in\_range(id))  
  
 *def* test\_bootstrap\_within\_bootstrapping\_bucket(*self*):  
 *"""  
 Test the bootstrap process within a bootstrapping bucket scenario.  
  
 This test creates a network with 22 virtual protocols representing nodes.  
 It sets up a bootstrap peer, with the bootstrapper having knowledge of 10 other nodes,  
 and one of those nodes having knowledge of another 10 nodes. The goal is to simulate  
 the bootstrap process and ensure that the expected number of contacts is received.  
  
 We need 22 virtual protocols. One for the bootstrap peer,  
 10 for the nodes the bootstrap peer knows about, and 10 for  
 the nodes one of the nodes knows about. And one for us to  
 rule them all.  
 :return: None  
 """* vp: list[VirtualProtocol] = []  
 *# creates 22 virtual protocols  
 for* i *in* range(22):  
 vp.append(VirtualProtocol())  
  
 *# us* dht\_us: DHT = DHT(ID.random\_id(), vp[0], storage\_factory=VirtualStorage, router=Router())  
 vp[0].node = dht\_us.\_router.node  
  
 *# our bootstrap peer* dht\_bootstrap: DHT = DHT(ID.random\_id(), vp[1], storage\_factory=VirtualStorage, router=Router())  
 vp[1].node = dht\_bootstrap.\_router.node  
  
 *# stops pycharm saying it could be undefined later on. THIS LINE IS USELESS.* n: Node = Node(Contact(ID.random\_id(), vp[0]), VirtualStorage())  
  
 *# Our bootstrapper knows 10 contacts  
 for* i *in* range(10):  
 c: Contact = Contact(ID.random\_id(), vp[i + 2])  
 n: Node = Node(c, VirtualStorage())  
 vp[i + 2].node = n  
 dht\_bootstrap.\_router.node.bucket\_list.add\_contact(c)  
  
 *# One of those nodes, in this case the last one we added to our bootstrapper (for convenience's sake)  
 # knows about 10 other contacts  
  
 # n is the last one our bootstrapper knows* node\_who\_knows\_10 = n *# Ignore PyCharm error saying it can be referenced before being created.  
 del* n *# bad naming, don't want to use it later on.  
  
 # create the 10 it knows about  
 for* i *in* range(10):  
 c: Contact = Contact(ID.random\_id(), vp[i + 12])  
 n2: Node = Node(c, VirtualStorage())  
 vp[i + 12].node = n2  
 node\_who\_knows\_10.bucket\_list.add\_contact(c)  
  
 dht\_us.bootstrap(dht\_bootstrap.\_router.node.our\_contact)  
  
 sum\_of\_contacts = len(dht\_us.\_router.node.bucket\_list.contacts())  
 print(f"sum of contacts: {sum\_of\_contacts}")  
 *self*.assertTrue(sum\_of\_contacts == 11,  
 "Expected our peer to get 11 contacts.")  
  
 *def* test\_bootstrap\_outside\_bootstrapping\_bucket(*self*):  
 *"""  
 Test the bootstrap process when bootstrapping from a DHT node with contacts outside its own bucket.  
  
 This test simulates the scenario where a DHT node (dht\_us) attempts to bootstrap from another node  
 (dht\_bootstrap) whose contact list includes nodes outside its immediate bucket range. The goal is to  
 verify that the bootstrapping process correctly adds and organizes contacts, and handles bucket splits.  
  
 Steps:  
 1. Create two DHT nodes, dht\_us and dht\_bootstrap, and establish virtual protocols (vp) for communication.  
 2. Populate dht\_bootstrap with 20 contacts, with one contact having an ID >= 2 \*\* 159 to trigger a bucket split.  
 3. Add 10 contacts to one of the nodes in dht\_bootstrap's contact list (simulating contacts outside the bucket).  
 4. Perform bootstrap operation from dht\_us to dht\_bootstrap.  
 5. Verify that dht\_us has a total of 31 contacts after bootstrapping.  
  
 Raises:  
 AssertionError: If the number of contacts in dht\_us after bootstrapping is not 31.  
 """* vp: list[VirtualProtocol] = []  
 *for* i *in* range(32):  
 vp.append(VirtualProtocol())  
  
 *# Us, ID doesn't matter.* dht\_us: DHT = DHT(ID.random\_id(), vp[0], storage\_factory=VirtualStorage, router=Router())  
 vp[0].node = dht\_us.\_router.node  
  
 *# our bootstrap peer  
 # all IDs are < 2 \*\* 159* dht\_bootstrap: DHT = DHT(ID.random\_id(0, 2 \*\* 159 - 1), vp[1], storage\_factory=VirtualStorage, router=Router())  
 vp[1].node = dht\_bootstrap.\_router.node  
 *# print(sum([len([c for c in b.contacts]) for b in dht\_bootstrap.\_router.node.bucket\_list.buckets]))  
  
 # Our bootstrapper knows 20 contacts  
 for* i *in* range(19):  
 *# creating 19 shell contacts* id: ID = ID.random\_id(0, 2 \*\* 159 - 1)  
 c: Contact = Contact(id, vp[i + 2])  
 c.protocol.node = Node(c, VirtualStorage())  
 dht\_bootstrap.\_router.node.bucket\_list.add\_contact(c)  
  
 *# for 20th  
 # all IDs are < 2 \*\* 159, except the last one, which is >= 2 \*\* 159  
 # Which will force a bucket split for us* id = ID.max()  
 important\_contact: Contact = Contact(id, vp[21])  
 n = Node(important\_contact, VirtualStorage())  
  
 *# add 10 contacts to node  
 # this basically means that the bootstrapper knows 20 contacts, one of them knows 10 contacts.  
 # we're trying to add all 30 + bootstrapper so 31.  
 for* i *in* range(10):  
 *# creating 10 shell contacts* c2: Contact = Contact(ID.random\_id(), vp[i + 22])  
 n2 = Node(c2, VirtualStorage())  
 vp[i + 22].node = n2  
 *# adding the 10 shell contacts* n.bucket\_list.add\_contact(c2) *# Note we're adding these contacts to the 10th node.* important\_contact.protocol.node = n  
 dht\_bootstrap.\_router.node.bucket\_list.add\_contact(important\_contact) *# adds the 1 important contact.  
  
 # just making sure vp[i + 2].node = n works retrospectively on c.  
  
 self*.assertTrue(n.our\_contact.id == important\_contact.protocol.node.our\_contact.id == ID.max())  
  
 *self*.assertTrue(len(n.bucket\_list.contacts()) == 10,  
 f"contacts: {len(n.bucket\_list.contacts())}")  
  
 *self*.assertTrue(len(important\_contact.protocol.node.bucket\_list.contacts()) == 10,  
 f"contacts: {len(n.bucket\_list.contacts())}")  
  
 *self*.assertTrue(important\_contact.id == ID.max(), "What else could it be?")  
  
 *self*.assertTrue(ID.max() *in* [c.id *for* c *in* dht\_bootstrap.\_router.node.bucket\_list.contacts()],  
 "Contact we just added to bucket list should be in bucket list.")  
  
 *# print("DHT Bootstrap contact length =", len(dht\_bootstrap.\_router.node.bucket\_list.contacts()))  
 self*.assertTrue(len(dht\_bootstrap.\_router.node.bucket\_list.contacts()) == 20,  
 "DHT Bootstrapper must have 20 contacts.")  
 *# One of those nodes, in this case specifically the last one we added to our bootstrapper so that it isn't in  
 # the bucket of our bootstrapper, we add 10 contacts. The IDs of those contacts don't matter.  
  
 self*.assertTrue([len(b.contacts) *for* b *in* n.bucket\_list.buckets] == [10],  
 "Must have 10 contacts in node.")  
  
 *# print("Starting bootstrap...")* dht\_us.bootstrap(dht\_bootstrap.\_router.node.our\_contact)  
 *# print("Bootstrap finished!")  
  
 # print(f"\nLength of buckets: {[len(b.contacts) for b in dht\_us.\_router.node.bucket\_list.buckets]}")* sum\_of\_contacts = len(dht\_us.\_router.node.bucket\_list.contacts())  
 *self*.assertTrue(sum\_of\_contacts == 31,  
 f"Expected our peer to have 31 contacts, {sum\_of\_contacts} were given.")  
  
  
*class* BucketManagementTests(unittest.TestCase):  
 *def* test\_non\_responding\_contact\_evicted(*self*):  
 *"""  
 Tests that a nonresponding contact is evicted after   
 Constants.EVICTION\_LIMIT tries.  
 """* dht = DHT(ID(0), VirtualProtocol(), storage\_factory=VirtualStorage, router=Router())  
 bucket\_list: BucketList = setup\_split\_failure(dht.node.bucket\_list)  
 *self*.assertTrue(len(bucket\_list.buckets) == 2,  
 "Bucket split should have occurred.")  
 *self*.assertTrue(len(bucket\_list.buckets[0].contacts) == 1,  
 "Expected 1 contact in bucket 0.")  
 *self*.assertTrue(len(bucket\_list.buckets[1].contacts) == 20,  
 "Expected 20 contacts in bucket 1.")  
  
 *# The bucket is now full. Pick the first contact, as it is last   
 # seen (they are added in chronological order.)* non\_responding\_contact: Contact = bucket\_list.buckets[1].contacts[0]  
 print(non\_responding\_contact.id)  
 *# Since the protocols are shared, we need to assign   
 # a unique protocol for this node for testing.* vp\_unresponding: VirtualProtocol = VirtualProtocol(  
 non\_responding\_contact.protocol.node,  
 *False* )  
 non\_responding\_contact.protocol = vp\_unresponding  
  
 next\_new\_contact = Contact(  
 ID(2 \*\* 159 + 1),  
 dht.our\_contact.protocol  
 )  
  
 *# Hit the nonresponding contact EVICTION\_LIMIT times  
 # Which will trigger the eviction algorithm.  
  
 for* \_ *in* range(Constants.EVICTION\_LIMIT):  
 bucket\_list.add\_contact(next\_new\_contact)  
  
 *self*.assertTrue(len(bucket\_list.buckets[1].contacts) == 20,  
 "Expected 20 contacts in bucket 1.")  
  
 *self*.assertTrue(len(bucket\_list.buckets[0].contacts) == 1,  
 f"Expected 1 contact in bucket 0, got {len(bucket\_list.buckets[0].contacts)}.")  
  
 *# Verify can\_split -> pending eviction happened  
 self*.assertTrue(len(dht.pending\_contacts) == 0,  
 "Pending contact list should now be empty.")  
 *self*.assertFalse(non\_responding\_contact *in* bucket\_list.contacts(),  
 "Expected bucket to NOT contain non-responding contact.")  
  
 *self*.assertTrue(next\_new\_contact *in* bucket\_list.contacts(),  
 "Expected bucket to contain new contact.")  
  
 *self*.assertTrue(len(dht.eviction\_count) == 0,  
 "Expected no contacts to be pending eviction.")  
  
 *def* test\_non\_responding\_contact\_delayed\_eviction(*self*):  
 *"""  
 Tests that a nonresponding contact puts the new contact into a pending list.  
 """* dht = DHT(ID(0), VirtualProtocol(), storage\_factory=VirtualStorage, router=Router())  
 bucket\_list: BucketList = setup\_split\_failure(dht.node.bucket\_list)  
  
 *self*.assertTrue(len(bucket\_list.buckets) == 2,  
 "Bucket split should have occurred.")  
  
 *self*.assertTrue(len(bucket\_list.buckets[0].contacts) == 1,  
 "Expected 1 contact in bucket 0.")  
  
 *self*.assertTrue(len(bucket\_list.buckets[1].contacts) == 20,  
 "Expected 20 contacts in bucket 1.")  
  
 *# The bucket is now full. pick the first contact,  
 # as it is last seen (they are added chronologically.)* non\_responding\_contact: Contact = bucket\_list.buckets[1].contacts[0]  
  
 *# Since the protocols are shared, we assign a unique protocol for this node for testing.* vp\_unresponding = VirtualProtocol(  
 node=non\_responding\_contact.protocol.node,  
 responds=*False* )  
 non\_responding\_contact.protocol = vp\_unresponding  
  
 *# set up the next new contact (it can respond.)* next\_new\_contact = Contact(  
 id=ID(2 \*\* 159 + 1),  
 protocol=dht.our\_contact.protocol  
 )  
 bucket\_list.add\_contact(next\_new\_contact)  
  
 *self*.assertTrue(len(bucket\_list.buckets[1].contacts) == 20,  
 f"Expecting 20 contacts in bucket 1, got {len(bucket\_list.buckets[0].contacts)}")  
  
 *self*.assertTrue(len(bucket\_list.buckets[0].contacts) == 1,  
 f"Expected 1 contact in bucket 0, got {len(bucket\_list.buckets[0].contacts)}")  
  
 *# Verify can\_split -> Evict happened.  
  
 self*.assertTrue(len(dht.pending\_contacts) == 1,  
 "Expected one pending contact.")  
 *self*.assertTrue(next\_new\_contact *in* dht.pending\_contacts,  
 "Expected pending contact to be the 21st contact.")  
 *self*.assertTrue(len(dht.eviction\_count) == 1,  
 "Expected one contact to be pending eviction.")  
  
  
*class* Chapter10Tests(unittest.TestCase):  
 *def* test\_new\_contact\_gets\_stored\_contacts(*self*):  
 *"""  
 Verify that we get stored values whose keys ^ contact ID are less than stored keys ^ other contacts.  
  
 There’s a lot of setup here for creating two existing contacts and two   
 key-values whose IDs have been specifically set. See the comments for   
 the XOR distance “math.”  
 """  
  
 # Set up a node at the midpoint  
 # The existing node haas the ID 10000...* existing: Node = Node(Contact(ID.mid(), *None*), VirtualStorage())  
 val\_1 = "Value 1"  
 val\_mid = "Value mid"  
  
 *# The existing node stores the 2 items, one with an ID "hash" of 1, the other with ID.max.  
 # Simple storage rather than executing the code for store.* existing.simply\_store(ID(1), val\_1)  
 existing.simply\_store(ID.mid(), val\_mid)  
  
 *self*.assertTrue(len(existing.storage.get\_keys()) == 2,  
 f"Expected the existing node to have 2 key-values. {existing.storage.get\_keys()}")  
  
 *# Create a contact in the existing node's bucket list that is closer to one of the values.  
 # This contact has the prefix 0100000...* other\_contact = Contact(ID(2 \*\* 158), *None*)  
 other = Node(other\_contact, VirtualStorage())  
 existing.bucket\_list.buckets[0].contacts.append(other\_contact)  
  
 *# The unseen contact has prefix 0110000...* unseen\_vp = VirtualProtocol()  
 unseen\_contact = Contact(ID(2 \*\* 158 + 2 \*\* 157), unseen\_vp)  
 unseen = Node(unseen\_contact, VirtualStorage())  
 unseen\_vp.node = unseen *# final fixup  
  
 self*.assertTrue(len(unseen.storage.get\_keys()) == 0, "The unseen node shouldn't have any key-values!")  
  
 *# An unseen node pings, and we should get back val\_min only as ID(1) ^ ID.mid() < ID.max() ^ ID.mid()  
  
 self*.assertTrue(ID(1) ^ ID.mid() < ID.max() ^ ID.mid(), (f"Fundamental issue with ID class. "  
 f"\n{ID(ID(1) ^ ID.mid()).bin()} \nvs "  
 f"\n{ID(ID.max() ^ ID.mid()).bin()}"))  
 existing.ping(unseen\_contact)  
  
 *# Contacts V1 V2  
 # 1000000 00...0001 10...0000  
 # 0100000  
 # maths:  
 # c1 ^ v1 c1 ^ v2 c2 ^ v1 c2 ^ v2  
 # 100...001 000...000 010...001 110...000  
 # c1 ^ v1 > c1 ^ v2, so v1 doesn't get send to the unseen node.  
 # c1 ^ v2 < c2 ^ v2, so it does get sent.  
  
 self*.assertTrue(  
 len(unseen.storage.get\_keys()) == 1,  
 "Expected 1 value stored in our new node."  
 )  
  
 *self*.assertTrue(  
 unseen.storage.contains(ID.mid()),  
 "Expected val\_mid to be stored."  
 )  
  
 *self*.assertTrue(  
 unseen.storage.get(ID.mid()) == val\_mid,  
 f"Expected val\_mid value to match, got {unseen.storage.get(ID.mid())}"  
 )  
  
  
*class* DHTSerialisationTests(unittest.TestCase):  
 *def* test\_serialisation(*self*):  
 dht: DHT = DHT(  
 id=ID.random\_id(),  
 protocol=VirtualProtocol(),  
 router=Router(),  
 storage\_factory=VirtualStorage  
 )  
 dht.save("kademlia/dht.pickle")  
  
 new\_dht = DHT.load("kademlia/dht.pickle")  
  
 *self*.assertTrue(  
 type(dht) == type(new\_dht),  
 f"Saved and loaded DHT are not the same type. {type(dht)} vs {type(new\_dht)}"  
 )  
 *self*.assertTrue(  
 dht.our\_id == new\_dht.our\_id,  
 "Saved and loaded DHT is not identical to the original."  
 )  
  
 *def* test\_circular\_serialisation(*self*):  
 dht: DHT = DHT(  
 id=ID.random\_id(),  
 protocol=VirtualProtocol(),  
 router=Router(),  
 storage\_factory=VirtualStorage  
 )  
  
 node = Node(  
 Contact(dht.our\_id),  
 storage=VirtualStorage()  
 )  
 dht.\_router.node = node  
  
 dht.save("kademlia/dht.pickle")  
  
 new\_dht = DHT.load("kademlia/dht.pickle")  
  
 *self*.assertTrue(  
 type(dht) == type(new\_dht),  
 "Saved and loaded DHT are not the same type. "  
 f"{type(dht)} vs {type(new\_dht)}"  
 )  
  
 *self*.assertTrue(  
 dht.\_router.node.our\_contact.id == new\_dht.\_router.node.our\_contact.id,  
 "Saved and loaded DHT is not identical to the original."  
 )  
  
 *def* second\_dht\_serialisation\_test(*self*):  
 p1: TCPSubnetProtocol = TCPSubnetProtocol(  
 "http://127.0.0.1/",  
 7124,  
 1  
 )  
  
 p2: TCPSubnetProtocol = TCPSubnetProtocol(  
 "http://127.0.0.1/",  
 7124,  
 1  
 )  
  
 store1: VirtualStorage = VirtualStorage()  
 store2: VirtualStorage = VirtualStorage()  
  
 *# Ensures that all nodes are closer, becuase ID.max() ^ n < ID.max()  
 # When n > 0* dht: DHT = DHT(  
 id=ID.max(),  
 protocol=p1,  
 router=Router(),  
 originator\_storage=store1,  
 republish\_storage=store1,  
 cache\_storage=VirtualStorage()  
 )  
  
 contact\_id: ID = ID.mid()  
 other\_contact: Contact = Contact(  
 id=contact\_id,  
 protocol=p2  
 )  
 other\_node: Node = Node(  
 contact=other\_contact,  
 storage=store2  
 )  
 *# Add this other contact to our peer list.* dht.node.bucket\_list.add\_contact(other\_contact)  
 dht.save(f"dht.{Constants.DHT\_SERIALISED\_SUFFIX}")  
  
 new\_dht: DHT = DHT.load(f"dht.{Constants.DHT\_SERIALISED\_SUFFIX}")  
 *self*.assertTrue(  
 new\_dht.node.bucket\_list.contacts() == 1,  
 "Expected our node to have 1 contact."  
 )  
 *self*.assertTrue(  
 new\_dht.node.bucket\_list.contact\_exists(other\_contact),  
 "Expected our contact to have the other contact."  
 )  
 *self*.assertTrue(  
 new\_dht.\_router.node == new\_dht.node,  
 "Router node not initialised."  
 )  
  
  
*class* TCPSubnetTests(unittest.TestCase):  
 @staticmethod  
 *def* setup():  
 local\_ip = "127.0.0.1"  
 valid\_server = *False* server = *None* port = 1  
 *while not* valid\_server:  
 port = random.randint(10000, 10500)  
 server = TCPSubnetServer(server\_address=(local\_ip, port))  
 valid\_server = *True* p1: TCPSubnetProtocol = TCPSubnetProtocol(url=local\_ip, port=port, subnet=1)  
 p2: TCPSubnetProtocol = TCPSubnetProtocol(url=local\_ip, port=port, subnet=2)  
  
 our\_id = ID.random\_id()  
  
 c1 = Contact(id=our\_id, protocol=p1)  
 c2 = Contact(id=ID.random\_id(), protocol=p2)  
  
 n1 = Node(c1, VirtualStorage())  
 n2 = Node(c2, VirtualStorage())  
  
 server.register\_protocol(p1.subnet, n1)  
 server.register\_protocol(p2.subnet, n2)  
 *# print(server.subnets)* thread = server.thread\_start()  
  
 *return* local\_ip, port, server, p1, p2, our\_id, c1, c2, n1, n2, thread  
  
 *def* test\_ping\_route(*self*):  
 *"""  
 Makes sure no exceptions are thrown when pinging a contact.  
 """* local\_ip, port, server, p1, p2, our\_id, c1, c2, n1, n2, thread = *self*.setup()  
  
 *# The actual test:* p2.ping(c1)  
  
 server.thread\_stop(thread)  
  
 *def* test\_store\_route(*self*):  
 local\_ip, port, server, p1, p2, our\_id, c1, c2, n1, n2, thread = *self*.setup()  
  
 *# The actual test:* sender = Contact(ID.random\_id(), p1)  
 test\_id: ID = ID.random\_id()  
 test\_value = "Test"  
 p2.store(sender, test\_id, test\_value)  
  
 *self*.assertTrue(n2.storage.contains(test\_id),  
 "Expected remote peer to have value.")  
 *self*.assertTrue(n2.storage.get(test\_id) == test\_value,  
 "Expected remote peer to contain stored value.")  
  
 server.thread\_stop(thread)  
  
 *def* test\_find\_nodes\_route(*self*):  
 local\_ip = "127.0.0.1"  
 valid\_server = *False* port = *None* server = *None  
 while not* valid\_server:  
 port = random.randint(10000, 10500)  
 server = TCPSubnetServer(server\_address=(local\_ip, port))  
 valid\_server = *True* p1 = TCPSubnetProtocol(url=local\_ip, port=port, subnet=1)  
 p2 = TCPSubnetProtocol(url=local\_ip, port=port, subnet=2)  
  
 our\_id = ID.random\_id()  
  
 c1 = Contact(id=our\_id, protocol=p1)  
 c2 = Contact(id=ID.random\_id(), protocol=p2)  
  
 n1 = Node(c1, VirtualStorage())  
 n2 = Node(c2, VirtualStorage())  
  
 *# Node 2 knows about another contact that isn't us  
 # - this is what we are trying to find* other\_peer = ID.random\_id()  
  
 n2.bucket\_list.buckets[0].contacts.append(  
 Contact(  
 other\_peer,  
 TCPSubnetProtocol(local\_ip, port, 3)  
 )  
 )  
 server.register\_protocol(p1.subnet, n1)  
 server.register\_protocol(p2.subnet, n2)  
 thread = server.thread\_start()  
  
 id = ID.random\_id()  
 ret, errors = p2.find\_node(c1, id)  
  
 *if* ret:  
 *self*.assertTrue(  
 len(ret) == 1,  
 f"Expected 1 contact, {len(ret)} were returned."  
 )  
  
 *self*.assertTrue(  
 ret[0].id == other\_peer,  
 "Expected contact to the other peer (not us).")  
 *else*:  
 *self*.assertTrue(  
 type(ret) == list[Contact],  
 "Expected find\_node to return 1 contact, 0 were returned."  
 )  
  
 server.thread\_stop(thread)  
  
 *def* test\_find\_value\_router(*self*):  
 local\_ip, port, server, p1, p2, our\_id, c1, c2, n1, n2, thread = *self*.setup()  
  
 *# Node 2 knows about another contact that isn't us  
 # - this is what we are trying to find* test\_id = ID.random\_id()  
 test\_value = "Test"  
 print("[Unit test] Store starting...")  
 p2.store(sender=c1, key=test\_id, val=test\_value)  
 print("[Unit test] Store done.")  
 *self*.assertTrue(  
 n2.storage.contains(test\_id),  
 "Expected remote peer to have value."  
 )  
  
 *self*.assertTrue(  
 n2.storage.get(test\_id) == test\_value,  
 "Expected node to store the correct value."  
 )  
  
 print("[Unit test] Find value starting...")  
 contacts, val, error = p2.find\_value(c1, test\_id)  
 print("[Unit test] Find value received:", contacts, val, error)  
 print("[Unit test] Find value done.")  
  
 *self*.assertFalse(  
 contacts, "Expected to find value." *# huh?* )  
 print(f"We stored '{val}' on the other node, we got back '{test\_value}'.")  
 *self*.assertTrue(  
 val == test\_value, "Value does not match expected value from peer."  
 )  
  
 *def* test\_unresponsive\_node(*self*):  
 local\_ip = "127.0.0.1"  
 port = 7124  
 server = TCPSubnetServer(server\_address=(local\_ip, port))  
  
 p1 = TCPSubnetProtocol(url=local\_ip, port=port, subnet=1)  
 p2 = TCPSubnetProtocol(url=local\_ip, port=port, subnet=2)  
 p2.responds = *False* our\_id = ID.random\_id()  
  
 c1 = Contact(id=our\_id, protocol=p1)  
 c2 = Contact(id=ID.random\_id(), protocol=p2)  
  
 n1 = Node(c1, VirtualStorage())  
 n2 = Node(c2, VirtualStorage())  
  
 server.register\_protocol(p1.subnet, n1)  
 server.register\_protocol(p2.subnet, n2)  
 thread = server.thread\_start()  
  
 test\_id = ID.random\_id()  
 test\_value = "Test"  
  
 error: RPCError = p2.store(c1, test\_id, test\_value)  
 *# print("[Unit tests] [Error]", error)  
 self*.assertTrue(  
 error.timeout\_error,  
 "Expected timeout when contacting unresponsive node."  
 )  
  
 server.thread\_stop(thread)  
  
  
*class* JSONStorageTests(unittest.TestCase):  
 *def* test\_get\_set(*self*):  
 *if* os.path.exists("1"):  
 shutil.rmtree("1")  
 storage = SecondaryJSONStorage(f"{ID(1)}/test\_storage.json")  
 store\_id = ID(1)  
 storage.set(store\_id, "Test")  
 *self*.assertTrue(storage.contains(store\_id), "Expected storage to contain data")  
 ret\_val = storage.get(store\_id)  
 *self*.assertEqual(ret\_val, "Test")  
  
 *def* test\_remove(*self*):  
 *if* os.path.exists("1"):  
 shutil.rmtree("1")  
 storage = SecondaryJSONStorage(f"{ID(1)}/test\_storage.json")  
 storage.set(ID(2), "to remove")  
 *self*.assertTrue(storage.contains(2), "We should have added the ID.")  
 storage.remove(2)  
 *self*.assertFalse(storage.contains(2), "Should have removed the ID.")  
  
  
*class* IDIntegerTests(unittest.TestCase):  
 *def* test\_xor(*self*):  
 id\_23 = ID(23)  
 *self*.assertTrue(ID(23) ^ 14 == 23 ^ 14) *# Typical  
 self*.assertTrue(ID(14) ^ 23 == 14 ^ 23) *# Typical  
 self*.assertTrue(ID(2352) ^ 53 == 2352 ^ 53) *# Typical  
 self*.assertTrue(ID(0) ^ 0 == 0 ^ 0) *# Boundary  
 self*.assertTrue(ID(2 \*\* 160 - 1) ^ 4 == (2 \*\* 160 - 1) ^ 4) *# Boundary  
  
 def* test\_ranges(*self*):  
 *with self*.assertRaises(ValueError):  
 overrange\_id = ID(2 \*\* 160) *# Boundary Erroneous  
  
 with self*.assertRaises(ValueError):  
 overrange\_id = ID(2 \*\* 160 + 7) *# Erroneous  
  
 with self*.assertRaises(ValueError):  
 overrange\_id = ID(-1) *# Erroneous  
  
 def* test\_equal(*self*):  
 *self*.assertTrue(ID(1) == 1)  
 *self*.assertTrue(ID(34) == 34)  
  
 *def* test\_lt(*self*):  
 *self*.assertTrue(ID(1) < 2)  
 *self*.assertTrue(ID(54) < 70)  
  
 *def* test\_le(*self*):  
 *self*.assertTrue(ID(1) <= 2)  
 *self*.assertTrue(ID(2) <= 2)  
 *self*.assertTrue(ID(54) <= 70)  
 *self*.assertTrue(ID(70) <= 70)  
  
 *def* test\_gt(*self*):  
 *self*.assertTrue(ID(1) >= 1)  
 *self*.assertTrue(ID(1) >= 0)  
 *self*.assertTrue(ID(100) >= 100)  
 *self*.assertTrue(ID(2 \*\* 160 - 1) >= 1)  
  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

# Test References

|  |  |
| --- | --- |
| Reference | Screenshots |
| 1 | “Copy Me!” is copied to clipboard. |
| 2 | “80085” copied to clipboard. |
| 3 | “<class 'int'>” copied to clipboard. |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 | I added this: |
| 10 | \_ |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |
| 16 |  |
| 17 |  |
| 18 |  |
| 19 |  |
| 20 |  |
| 21 |  |
| 22 |  |
| 23 | I had forgotten to move the peer once its in the network to the main network screen, after the fix: |
| 24 |  |
| 25 | I had my Regex match statement the wrong way round. |
| 26 | ^ When contacting contact that does not exist. |
| 27 | I missed a greater than or equal to.    It correctly is identified as an error now.    Validation passed, meaning 2^160 – 1 is valid, as it should be. |

# THE END.

